

Spatial Memory Behaviour of the Tower of Hanoi Algorithm on the Java Virtual Machine: An Analytical and Empirical Study

Omar Khan Durrani, Almas Sadaf Farooqi, Harshitha Sindhe

Dept. of Computer Science & Engineering Ghousia College of Engineering, Ramanagram

Abstract- Asymptotic analysis of recursive algorithms usually emphasises time complexity, leaving space underexamined on managed runtimes, where realised memory cost depends on object layout and collector behaviour as well as algorithmic structure. This study uses the Tower of Hanoi (TOH), whose optimal solution comprises $2^N - 1$ moves, as a controlled workload for characterising spatial performance in Java. Three paradigms are formalised: (i) pure recursion, with auxiliary space bounded by call-stack depth, $O(N)$; (ii) an iterative variant with an explicit stack, relocating the same $O(N)$ state to the heap; and (iii) a move-materialising variant retaining the full sequence, requiring $\Theta(2^N)$ space. Analytical footprint models are derived from frame size, object-header size, reference width, and alignment padding. The models are validated using the Java Runtime API, sampling live-heap occupancy after forced collection across increasing N until resource exhaustion. Repeated trials yield mean estimates, and model fit is assessed by regression on log-transformed data. Sensitivity to heap bounds, thread stack size, and collector choice is analysed to separate algorithmic effects from runtime effects. Asymptotic space classes are preserved on the JVM, but materialisation exhausts the heap at far smaller N than recursion, while deep recursion fails through `StackOverflowError`. Lazy move generation is recommended.

Keywords— Tower of Hanoi, Spatial Complexity, Heap Memory, Java Virtual Machine, Recursion, Garbage Collection, Algorithm Performance

I. INTRODUCTION

Algorithm analysis characterises the time and space a procedure requires as a function of the input size n , using step counts or recurrence relations and asymptotic notation [1], [2]. These theoretical predictions are then validated by implementing the algorithm and measuring it across a range of input sizes. The Tower of Hanoi (TOH), proposed by Édouard Lucas in 1883 [3], is a canonical subject for this approach.

It requires N disks of different sizes to be moved between three rods without ever placing a larger disk on a smaller one. The optimal solution is recursive and satisfies $T(N) = 2T(N-1) + 1$ with $T(1) = 1$, giving $T(N) = 2^N - 1$ moves and a time complexity of $\Theta(2^N)$ [2]. Because the move count is fixed for a given N , TOH has no separate best, average, and worst cases [1], and its exponential growth is visible even at small N .

Among the many languages in use, C++, Java, and Python are the most widely adopted: Python, C, C++, and Java held the top four places in the TIOBE index in September 2026 [4]. This paper therefore refers to C++, Java, and Python as popular programming languages (PPLs); their development is traced in Section II-A. They differ markedly in memory management. C++ leaves allocation and release largely to the programmer [5], Java relies on a garbage-collected heap managed by the JVM [6], and the reference implementation of Python combines reference counting with a cyclic garbage collector [7]. These differences make memory behaviour a natural basis for comparing PPLs, and they motivate a precise, language-specific analysis of space, which this study carries out for Java.

While this time behaviour is well documented, the memory behaviour of TOH on managed runtimes has received little systematic study. In a language-neutral model, a purely recursive solution needs $O(N)$ auxiliary space, because recursion depth

cannot exceed the number of disks [2], whereas storing every move requires $\Theta(2^N)$. In Java the difference is more than asymptotic. Each Java Virtual Machine (JVM) thread owns a call stack that limits recursion depth, while all run-time objects reside on a shared heap managed by the garbage collector [6]. The recursive version can therefore fail only through `StackOverflowError`, and the move-storing version through `OutOfMemoryError`, two failures with different causes and different JVM controls. Constant factors also matter: object headers, alignment padding, reference width, `ArrayList` resizing, and collector behaviour [8] can make two implementations of the same asymptotic class differ markedly in their practical limits.

TOH is widely used as a workload in cross-language performance studies [9], [10]. Durrani et al. [9] attributed Java's faster TOH execution relative to C++ mainly to its lower memory use, yet they neither measured nor modelled that use. Evidence from other work shows why measurement is needed: studies of sorting algorithms in Java and Python, two of today's most popular programming languages, have reported results that do not always follow the theoretical representation [11], [12]. The question is also practical. Containerised deployments on Docker and Kubernetes impose hard memory limits through Linux control groups [13], [14], so a program that is acceptable in time but excessive in space may be terminated by the operating system (an OOMKill) even when it runs correctly in unrestricted testing. This study therefore examines the spatial behaviour of TOH in Java, and its findings are expected to apply to any recursive program that allocates many small heap objects.

The aim of this study is to establish how closely measured memory consumption follows the theoretical models in Java. Four objectives support this aim: (i) to derive analytical stack and heap models for each TOH implementation; (ii) to quantify how much of the per-move cost comes from JVM object overhead and container growth rather than from the stored data; (iii) to determine how heap size, thread stack size, and collector choice change the largest problem size that can be solved; and (iv)

to translate these findings into guidance for memory-efficient implementation.

To meet these objectives, three implementations are compared under identical conditions: a pure recursive version, an iterative version with an explicit stack, and a version that records every move in a dynamic collection. The memory footprint of each is first derived analytically and is then measured with the Java Runtime API after forced garbage collection, so that live-heap occupancy is separated from short-lived allocation. The number of disks is increased from $N = 5$ to $N = 25$, and each configuration is repeated to obtain stable mean values.

The analysis is limited to single-threaded Java implementations and to the memory consumed by the algorithm's own data structures; execution time is reported only where it helps to interpret memory behaviour. Results are specific to the JVM versions and settings used, although the asymptotic conclusions are expected to hold more generally. The study matters for two reasons. It links algorithm theory with runtime engineering by showing where asymptotic classes stop predicting practical limits, and it gives developers a basis for sizing memory in constrained deployments. The same methodology can also be applied to other exponential-time recursive algorithms.

The main contributions of this paper are as follows:

- A formal derivation of the stack and heap space complexity of recursive, explicit-stack, and state-recording TOH implementations.
- A reproducible Java benchmark that measures heap consumption through the Runtime API with controls for garbage-collection noise, for $N = 5$ to $N = 25$.
- An analysis of how JVM object-header overhead and `ArrayList` growth affect the memory cost of each stored move.
- A comparative framework and practical JVM tuning guidance for memory-efficient deployment.

The rest of the paper is organised as follows. Section II presents the background and related work, Section III covers algorithm design and complexity analysis,

Section IV describes the benchmark methodology, Section V presents and discusses the results, Section VI relates them to the JVM memory architecture, Section VII sets out the limitations, and Section VIII concludes with directions for future work.

II. BACKGROUND AND RELATED WORK

1. Evolution of Programming Languages and Popular Programming Languages

A computer executes a program, a set of instructions that is first held in memory and then carried out by the processor. The earliest programs were written in binary machine code, the native language of the machine, and demanded highly skilled programmers. Assembly languages eased the task by replacing binary patterns with mnemonic symbols that are easier to remember and recognise. As computing proved its value in science, engineering, and commerce, compilers and interpreters were built to translate programs written in readable notation, and high-level languages such as FORTRAN, COBOL, BASIC, and Pascal appeared, making problem solving by computer far more accessible [15]. These stages are conventionally described as the first three generations of programming languages.

Growth in application size brought rising code volume and growing difficulty in debugging and maintenance. The 1968 NATO conference on software engineering [16] gave the name software crisis to these problems, which included projects that exceeded their budgets and schedules and software that was of low quality and hard to maintain. The C language, developed at Bell Laboratories in the early 1970s by Dennis Ritchie and documented by Kernighan and Ritchie [17], [18], addressed part of the problem. It was a high-level language that still gave close control of the machine, its rich data types and libraries made it attractive, and it was used to implement operating systems. As software for larger projects was demanded and the weaknesses of procedural decomposition became apparent, object-oriented technology gained ground. Smalltalk introduced the object-oriented model in the early 1970s [19], and C++, created by Bjarne Stroustrup in the 1980s as an extension of C, brought classes and related object-oriented concepts to

systems programming [5]. In parallel, researchers such as Hoare [20] argued for principled approaches to language design, and such work influenced later language and compiler designers.

Java was conceived at Sun Microsystems from 1991 by James Gosling and colleagues and was released in 1995 [21]. It combined an object-oriented model with built-in multithreading and network programming support, and it executes on the JVM. Python was created by Guido van Rossum at CWI in the Netherlands [22]; it emphasises simplicity and readability and has been widely adopted. Together with C and C++, Java and Python have become the most popular programming languages, and the TIOBE index placed Python, C, C++, and Java in the top four positions in September 2026 [4]. On this evidence, C++, Java, and Python are referred to in this paper as popular programming languages (PPLs).

Table I summarises how the three PPLs manage memory and how they fail when it is exhausted [5]–[7], [23]. The differences mean that a space analysis cannot simply be transferred from one language to another: the same algorithm places its state on a call stack or a heap under different rules, and it fails through different mechanisms. This is why a language-specific treatment of space, such as the one developed here for Java, is needed.

Table 1: Memory Management and Space-Related Failures in Popular Programming Languages

Language	Typical memory management	Deep recursion fails through	Heap exhaustion fails through
C++	Manual allocation and release; smart pointers and RAII	Stack overflow (undefined behaviour, usually a crash)	std::bad_alloc
Java	Garbage-collected heap managed by the JVM	StackOverflowError or	OutOfMemoryError
Python (CPython)	Reference counting with a cyclic garbage collector	RecursionError at the configured recursion limit	MemoryError

2. The Tower of Hanoi: Formulation and Known Results

Since its introduction by Lucas [3], the Tower of Hanoi has been studied extensively as a mathematical object, and Hinz et al. [24] give a comprehensive account of its history, variants, and analysis. The classical recursive solution yields the recurrence $T(N) = 2T(N-1) + 1$, whose solution, $2^N - 1$, is the minimum possible number of moves [2], [24]. The literature then develops in several directions. Iterative formulations that avoid recursion and generate the move sequence directly were proposed early [25], [26], and generalisations and variants, including parallel moves, have been analysed for optimal move counts [27]. In the works cited here, the emphasis is on the number and structure of moves, with time as the principal cost. Where space is discussed, it is treated analytically, as in the $O(N)$ and $\Theta(2^N)$ models of Section I, and not measured on a concrete runtime.

3. Space Complexity on the Java Virtual Machine

The JVM specification defines a private stack of frames for each thread and a heap shared by all threads [6], so the same logical state has a different cost depending on where it resides. Frame size depends on the local variables and operand stack of each method, whereas object size depends on the header, the fields, and alignment padding. Tools such as the OpenJDK Java Object Layout (JOL) project report the actual layout of objects [28], which makes analytical footprint models directly checkable. Garbage collection adds a further dependence: its cost varies with the size of the live set and the heap headroom available [8]. Hertz and Berger [29] showed that the trade-off between memory and speed in garbage-collected systems depends strongly on heap size relative to live data. A space analysis for Java that reports only the asymptotic class therefore omits the constants that determine practical limits.

4. Measurement Methodology on Managed Runtimes

Time and memory measurements on the JVM are affected by just-in-time compilation, the timing of garbage collection, and other sources of non-determinism, so a single run can mislead. Georges et

al. [30] observed that many Java performance studies lacked statistical rigour and proposed a methodology based on repeated invocations and confidence intervals. The DaCapo suite [31] was developed to provide realistic Java workloads together with sound benchmarking practice for managed runtimes. These works motivate the choices made in this study of forced collection before sampling, repeated trials, and explicit reporting of variability.

5. Cross-Language Performance Studies

Prechelt [32] compared seven languages on a common task and reported differences between languages in run time and memory consumption. Pereira et al. [33] measured run time, memory use, and energy consumption across 27 languages on ten programming problems, and found that memory use influences energy consumption. Studies centred on recursive problems are closer to the present work. Cubukcu et al. [10] compared C, Java, Python, and Matlab on Fibonacci and Tower of Hanoi implementations in terms of code length, programming effort, runtime efficiency, and reliability. Durrani et al. [9] compared Java and C++ on TOH and attributed Java's execution-time advantage to lower memory consumption. Studies of sorting algorithms in Java and Python [11], [12] further suggest that theoretical predictions and measured behaviour do not always agree.

6. Research Gap

In summary, prior work establishes the move structure and time complexity of TOH [2], [24], defines the JVM memory model [6], and supplies methodology for measuring Java programs [30], [31]. To the authors' knowledge, however, no study isolates the stack and heap footprint of TOH variants on the JVM, relates it to an analytical model that includes object-level constants, and quantifies how JVM parameters shift the practical limits. Durrani et al. [9] point to memory as a decisive factor but do not measure it. The present study addresses this gap.

III. ALGORITHM DESIGN AND COMPLEXITY ANALYSIS

1. Algorithm Design and Analysis Framework

Algorithm design is the selection of a strategy for transforming input into output, and the common strategies are brute force, divide and conquer, greedy methods, dynamic programming, and backtracking [1], [2]. TOH is a textbook case of divide and conquer: a problem of size N is reduced to two problems of size $N-1$ and one elementary step. Analysis then measures the resources that the chosen design consumes. For time, the analyst fixes a basic operation, here the movement of one disk, and counts how often it executes as a function of the input size. For recursive algorithms this count is expressed as a recurrence relation and solved in closed form [1], [2]. Space is treated in the same way, by counting the memory used beyond the input. Because the cost of many algorithms varies with the input, results are usually reported for the best, average, and worst cases.

Growth rates are compared with asymptotic notation. A function $f(n)$ is $O(g(n))$ if it is bounded above by a constant multiple of $g(n)$ for all sufficiently large n , $\Omega(g(n))$ if it is bounded below in the same way, and $\Theta(g(n))$ if both bounds hold [2]. Common order-of-growth classes are logarithmic, linear, $n \log n$, quadratic, and exponential. An exponential cost such as 2^n doubles with every unit increase in n , which is the behaviour examined in this paper.

2. The Recursive Algorithm

The canonical solution is a three-step divide-and-conquer procedure. To move N disks from a source rod S to a destination rod D using an auxiliary rod A , the algorithm (i) recursively moves $N-1$ disks from S to A , using D as the auxiliary rod; (ii) moves the N th, largest disk from S to D ; and (iii) recursively moves the $N-1$ disks from A to D , using S as the auxiliary rod. Steps (i) and (iii) are instances of the same problem with one disk fewer, and step (ii) is the only elementary operation.

Listing 1 gives this solution in Java together with the move-recording variant studied in this paper. The

method `hanoi` moves n disks from rod `from` to rod `to`, using rod `aux` as the intermediate. The variant `hanoiRecord` performs the same moves but keeps each one as an object in a list, so that the full solution remains available after the call returns. The two methods have the same control structure and differ only in what happens at the elementary step.

```
static long moves;

static void hanoi(int n, int from, int to, int aux) {
    if (n == 0) return;
    hanoi(n - 1, from, aux, to);
    moves++; // move disk n: from
    -> to
    hanoi(n - 1, aux, to, from);
}

record Move(int disk, int from, int to) {}

static void hanoiRecord(int n, int from, int to, int aux,
    List<Move> log) {
    if (n == 0) return;
    hanoiRecord(n - 1, from, aux, to, log);
    log.add(new Move(n, from, to)); // retained on the
    heap
    hanoiRecord(n - 1, aux, to, from, log);
}
```

Listing 1. Pure recursive and move-recording Tower of Hanoi in Java

3. Recurrence Derivation and Time Complexity

Let $T(N)$ be the number of disk moves performed for N disks. Steps (i) and (iii) each cost $T(N-1)$ and step (ii) costs one move, so

$$T(N) = 2T(N - 1) + 1, T(0) = 0. \quad (1)$$

The recurrence is solved by successive substitution:

$$\begin{aligned} T(N) &= 2T(N - 1) + 1 = 2[2T(N - 2) + 1] + 1 = 4T(N - 2) + 3 \\ &= \dots = 2^k T(N - k) + (2^k - 1). \end{aligned} \quad (2)$$

Setting $k = N$ and using the base case $T(0) = 0$ gives

$$T(N) = 2^N - 1. \quad (3)$$

The recursion can be viewed as a complete binary tree of calls. If the base case $N = 0$ is counted as a call, the tree has $2^N - 1$ internal nodes, each performing exactly one disk move, and 2^N leaves, which are the base cases. The total number of calls is therefore

$$C(N) = (2^N - 1) + 2^N = 2^{N+1} - 1, \quad (4)$$

in agreement with the recurrence $C(N) = 2C(N - 1) + 1$ and $C(0) = 1$. The moves are performed in the order in which the internal nodes are visited in an in-order traversal of this tree. The equality between internal nodes and moves is what links time to space: if every move is stored, the heap must hold one object per internal node, $2^N - 1$ in all (Section III-D).

Each call executes a bounded number of instructions, so the running time is $\Theta(2^N)$. Because the same number of moves is performed for every instance of size N , TOH has no distinct best, average, and worst cases. Adding one disk doubles the work: $N = 5$ requires 31 moves, whereas $N = 25$ requires 33,554,431.

4. Space Complexity: Two Spatial Regimes

Two spatial regimes arise, depending on whether the move history is retained. Table II summarises the implementations discussed below.

Regime 1: pure recursion (stack-dominant). Moves are printed or discarded as they are generated, so the only memory that grows with N is the call stack, which holds one frame for each pending call along a single path from the root of the call tree. The depth $D(N)$ satisfies $D(N) = D(N - 1) + 1$ with $D(0) = 1$, so

This is also the minimum for any legal solution. The largest disk must move at least once, all $N-1$ smaller disks must stand on the third rod when it does so, and they must afterwards be moved onto it, so any solution satisfies $M(N) \geq 2M(N-1) + 1$ [24].

where f is the size of one frame. The depth is N when the base case is tested before the recursive call is made, and $N + 1$ when the call with $n = 0$ is counted, as in Listing 1; in both cases the auxiliary space is $\Theta(N)$. This is the regime of the pure recursive implementation reported by Durrani et al. [9], and it is consistent with the low memory footprint observed there for Java. The frame size f is a runtime constant that depends on the local variables and operand stack of the method and on whether the just-in-time compiler has compiled it [6]. Because time grows as 2^N , N can never be large in practice: even at 10^9 moves per second, $N = 60$ would take more than three decades. The recursion depth thus

stays below a few dozen frames, far below the thousands of frames that a default thread stack accommodates, and a `StackOverflowError` is unreachable for this implementation unless the thread stack size (`-Xss`) is deliberately reduced.

Regime 2: state recording (heap-dominant). Each of the $2^N - 1$ moves is stored as a heap-allocated object, so the auxiliary space is $\Theta(2^N)$ and the total consumption is $S(N) = (2^N - 1) \cdot c$, where c is the cost of one recorded move in bytes (derived in Section III-E). The backing array of a dynamic list such as `ArrayList` grows geometrically, which keeps the amortised cost of an insertion constant but means that the old and new arrays coexist while the contents are copied. The peak footprint therefore exceeds the final live size.

Iterative variants. An iterative version with an explicit stack replaces the call stack by a stack of frame objects on the heap, each holding n , the three rods, and a resume point. At most $N + 1$ such objects exist at once, so the pending work needs only $\Theta(N)$ heap, charged with object headers and padding [28], but the heap still grows as $\Theta(2^N)$ if the full history is retained. A bit-manipulation iteration, in the family of iterative formulations that avoid recursion [25], [26], determines each move directly from a move counter and needs only a constant number of variables, so its auxiliary space is $O(1)$. The price is code that is harder to read and to relate to the recursive definition. It is included in Table II as a reference point and is not benchmarked in this study.

Table 2: Time and Space Complexity of the Toh Implementations

Implementation	Time	Aux. space	Location	Space-related failure
Pure recursion	$\Theta(2^N)$	$\Theta(N)$	Call stack	<code>StackOverflowError</code> (only if the stack is very small)

Explicit stack	$\Theta(2^N)$	$\Theta(N)$	Heap	OutOfMemory Error (not reachable in practice)
Move recording	$\Theta(2^N)$	$\Theta(2^N)$	Heap	OutOfMemory Error at moderate N
Bit-manipulation iteration (reference only)	$\Theta(2^N)$	$O(1)$	Local variables	None
Cyclic iteration (reference)	$\Theta(2^N)$	$O(1)$	Locals	None

5. Analytical Memory Model

The asymptotic classes hide constants that decide practical limits, so a concrete model is needed. For the move-recording variant let o be the size of one Move object, r the width of a reference, and a the fixed cost and slack of the backing array. Then

$$S_{\text{heap}}(N) = (2^N - 1) \cdot (o + r) + a. \quad (6)$$

The value of o follows from the object layout [28]. On a 64-bit HotSpot JVM with compressed object and class pointers, an object header occupies 12 bytes, made up of an 8-byte mark word and a 4-byte compressed class pointer. A Move with an int disk number and two char rod labels carries 8 bytes of field data, and the Move of Listing 1 with three int fields carries 12 bytes. Both objects occupy 24 bytes once rounded up to the 8-byte alignment unit, and the first layout wastes 4 bytes of padding.

A compressed reference occupies 4 bytes, so each recorded move costs about 28 bytes and $S_{\text{heap}}(N) \approx 28(2^N - 1)$ bytes. Of these 28 bytes only 12 hold information, and the header alone accounts for about 43 percent. Storing the same move as a packed int, with 2 bits for each rod and 5 bits for the disk number, which suffices for $N \leq 25$, needs 4 bytes, a reduction by a factor of seven. A short array would halve this again.

Table 3: gives the values predicted by Eq. (6) for the disk counts used in this study. They are model

predictions under the stated layout assumptions and are compared with measurements in Section V. They also show why the limit is practical rather than theoretical: at $N = 25$ the live set alone is about 896 MiB, and the transient copy during list growth adds more. Because the default maximum heap of HotSpot is a fraction of physical memory, a host with limited RAM can reach its heap limit near this size, whereas the recursive version needs only 26 frames.

Table 3: Predicted Live Heap for Move Objects (28 Bytes Per Move) and for a Packed Int Encoding (4 Bytes Per Move)

N	Moves ($2^N - 1$)	Stack frames ($N + 1$)	Move objects, 28 B/move	Packed ints, 4 B/move
5	31	6	868 B	124 B
10	1,023	11	≈ 28 KiB	≈ 4 KiB
15	32,767	16	≈ 896 KiB	≈ 128 KiB
20	1,048,575	21	≈ 28 MiB	≈ 4 MiB
25	33,554,431	26	≈ 896 MiB	≈ 128 MiB

6. Implications for the Study

Three conclusions follow from the analysis and frame the experiments. First, the time cost is identical for all implementations, so any difference in observed behaviour is a space effect. Second, pure recursion is safe in space for every feasible N , so the interesting comparison is between stack-resident and heap-resident state. Third, the move-recording variant is dominated by constants, namely object header, reference width, and container growth, which can be reduced by encoding moves compactly or by generating them lazily and streaming them to the consumer, the limiting case being the constant-space iteration [8].

IV. BENCHMARK METHODOLOGY

1. Experimental Design

The benchmark is a controlled experiment designed to test the models of Section III. It measures the memory retained by each implementation as the number of disks grows, and it determines how JVM settings change the practical limits. Table IV lists the factors and their levels. Four implementations are compared: pure recursion, the explicit-stack

iteration, move recording with Move objects, and a packed-int recording variant that quantifies the saving predicted in Table III. The disk count N runs from 5 to 25 in unit steps, and for each heap size further runs with N above 25 are made until an OutOfMemoryError occurs, which gives the exhaustion threshold. The recording variants are run both with default list growth and with the list pre-sized to $2^N - 1$ elements, which separates the cost of the objects from the cost of container growth. Maximum heap size, thread stack size, and collector are varied one at a time around a baseline configuration, so that each effect can be attributed to a single factor.

Table 4: Experimental Factors, Levels, and Purpose

Factor	Levels	Purpose
Implementation	Pure recursion; explicit stack; move recording (Move objects); packed-int recording	Compare stack-resident and heap-resident state
Disk count N	5 to 25 in unit steps; above 25 for exhaustion runs	Test $O(N)$ against $\Theta(2^N)$ growth
List capacity (recording)	Default growth; pre-sized to $2^N - 1$	Separate object cost from container growth
Maximum heap (-Xmx)	«levels, for example three heap sizes»	Locate the exhaustion threshold
Thread stack (-Xss)	«levels, for example three stack sizes»	Estimate frame size and recursion limit
Garbage collector	G1 (-XX:+UseG1GC); Parallel (-XX:+UseParallelGC); Serial (-XX:+UseSerialGC)	Effect on peak footprint and headroom

2. Implementations Under Test

The pure recursive version is the method hanoi of Listing 1. Its move counter is read after every run so that the just-in-time compiler cannot eliminate the computation as dead code. The explicit-stack version keeps pending work in an ArrayDeque of frame objects holding n, the three rods, and a resume point. The move-recording version is hanoiRecord of

Listing 1 and stores one Move object per move in an ArrayList. The packed-int version stores each move in one element of an int array, using 2 bits for each of the source and destination rods and 5 bits for the disk number, and the array is allocated once with exactly $2^N - 1$ elements. All versions run on a single thread and produce the same sequence of moves.

The harness of Listing 3 stores each move as a Move object in a growable ArrayList, which reproduces the state-recording regime and gives a measurable heap footprint. The list is created with the default capacity of 10 and grows by a factor of 1.5 at each resize. At its peak for $N = 20$, the backing array may hold up to about 1.57 million reference slots (4 bytes each under compressed ordinary object pointers), which adds roughly 6 MB of reference-array overhead beyond the Move objects themselves.

```
import java.util.ArrayList;
import java.util.List;

public class TowerOfHanoiMemoryBenchmark {
    static class Move {
        int disk; char from; char to;
        Move(int d, char f, char t) {
            disk = d; from = f; to = t;
        }
    }
    public static void main(String[] args) {
        int numDisks = 20;
        runGC();
        long memB = Runtime.getRuntime().totalMemory()
            - Runtime.getRuntime().freeMemory();
        long t0 = System.nanoTime();
        List<Move> log = new ArrayList<>();
        solveHanoi(numDisks, 'A', 'C', 'B', log);
        long t1 = System.nanoTime();
        long memA = Runtime.getRuntime().totalMemory()
            - Runtime.getRuntime().freeMemory();
        System.out.printf(
            "Moves:%d Time:%.2fms Heap:%.2fKB\n",
            log.size(), (t1 - t0) / 1e6,
            (memA - memB) / 1024.0);
    }
    static void solveHanoi(int n, char f,
        char t, char a, List<Move> s) {
        if (n == 0) return;
        solveHanoi(n - 1, f, a, t, s);
        s.add(new Move(n, f, t));
        solveHanoi(n - 1, a, t, f, s);
    }
    static void runGC() {
        for (int i = 0; i < 4; i++) {
            System.gc();
            try { Thread.sleep(100); }
            catch (InterruptedException e) { }
        }
    }
}
```

Listing 3. Self-contained benchmark harness for the move-recording variant

3. Test Environment

The experiments were run on [CPU model, number of cores, and clock speed] with 16 GB of memory

under Ubuntu 22.04 LTS. The primary JVM was OpenJDK 17.0.9, the HotSpot virtual machine in 64-bit mode with compressed ordinary object pointers enabled, which is the default for heaps below about 32 GiB. The experiments were repeated on a second environment, OpenJDK 21.0.10 on Ubuntu 24.04 LTS (Section V-E), to check that the exponential trend does not depend on the JVM. The baseline configuration used `-Xmx256m` for $N \leq 22$, `-Xmx2g` for $N = 25$, [baseline `-Xss`], and the Serial collector (`-XX:+UseSerialGC`), with the initial heap set equal to the maximum so that heap resizing does not occur during a run. Every combination of factors was run in a fresh JVM process started by a driver script, so that no state is carried from one run to the next.

4. Measurement Procedure

Memory is measured with the Java Runtime API, in which `totalMemory` reports the current heap size, `freeMemory` the unused part of it, and `maxMemory` the heap limit. Used heap is the difference between the first two. Because used heap includes garbage that has not yet been collected, the harness forces collection repeatedly until the reading stops decreasing (Listing 2). The retained heap of a run is the settled reading after the run minus the settled reading before it, taken while the result list is still reachable. Dividing it by $2^N - 1$ gives the measured bytes per move, which is compared with the 28 bytes predicted by the model.

```
static long settledHeap() throws InterruptedException {
    Runtime rt = Runtime.getRuntime();
    long previous = Long.MAX_VALUE;
    long used = rt.totalMemory() - rt.freeMemory();
    for (int i = 0; i < 10 && used < previous; i++) {
        previous = used;
        System.gc();
        Thread.sleep(50);
        used = rt.totalMemory() - rt.freeMemory();
    }
    return used;
}

long before = settledHeap();
List<Move> log = new ArrayList<>();
hanoiRecord(n, 0, 2, 1, log);
long after = settledHeap();
Reference.reachabilityFence(log); // log is still
reachable here
long retained = after - before;
```

Listing 2. Measurement of retained heap with the Runtime API

Accurate JVM heap measurement requires control of garbage-collection timing, just-in-time (JIT) warm-

up, and class-loading effects. The benchmark therefore adopts the following design, informed by [30], [31]:

- Warm-up: A single run at $N = 5$ is executed before measurement to trigger JIT compilation of the core method.
- Forced GC baseline: `System.gc()` is called four times, with 100 ms pauses between calls, immediately before the pre-execution heap baseline is recorded, which substantially increases the probability of a stable measurement.
- Runtime API measurement: Heap usage is computed as `Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()` immediately before and after algorithm execution. The difference is the net heap allocation attributable to the algorithm.
- Timing: Wall-clock time is captured with `System.nanoTime()` for comparison with the findings of [9].
- Multiple trials: Five trials are run for each disk count, and median values are reported.

The Serial collector (`-XX:+UseSerialGC`) is used throughout so that no concurrent collector threads can reclaim objects between the completion of the algorithm and the post-execution measurement.

Three supplementary facilities capture quantities that the Runtime API cannot. Garbage-collection logging (`-Xlog:gc`) and the garbage-collector management beans give collection counts and pause times. The memory-pool management beans give peak usage, including the transient copies made while a list grows. The Java Object Layout tool [28] confirms the layout and size of the Move object on the JVM under test.

The call stack is not part of the heap, so it is measured separately. The recursive method is run in a thread created with an explicit stack size, and the recursion depth is increased until a `StackOverflowError` is thrown. The frame size is then estimated

as the stack size divided by the largest depth reached, $f \approx \text{stack size} / \text{depth}$, both for interpreted code and after the method has been compiled [6]. Execution time is recorded only to help interpret the memory results, and each timed run is preceded by the warm-up run described above so that compilation has settled [30], [31].

V. STATISTICAL TREATMENT AND MODEL VALIDATION

Each configuration is repeated five times, each repetition in a separate JVM invocation, because a single run can mislead on a runtime with just-in-time compilation and non-deterministic collection [30]. Results are reported as the median with the sample standard deviation and a 95 percent confidence interval based on Student's t distribution. To validate the model, the logarithm to base 2 of the median retained heap is regressed on N by ordinary least squares. For $\Theta(2^N)$ growth the slope should be close to 1, and the intercept should estimate the logarithm of the per-move cost. The slope with its confidence interval and the coefficient of determination are reported. Agreement between prediction and measurement is expressed as the relative error (measured – predicted) / predicted. For the pure recursive implementation the corresponding test is that the retained heap shows no growth with N beyond measurement noise.

Threats to Validity and Reproducibility

Several limitations apply. A call to System.gc() is a request and not a guarantee, and freeMemory is an approximation, particularly with concurrent collectors and thread-local allocation buffers. These effects are reduced by the settling loop, by a fixed heap size, and by repeating the experiments under three collectors. The results are specific to the hardware and JVM versions used. Newer JDK releases offer an option for smaller object headers, which would change the constants o and r in Eq. (6) but not the asymptotic class, and the model is parameterised so that it can be re-evaluated for such layouts. Retained heap and peak heap are different quantities and both are reported, so that conclusions about live data are not confused with conclusions about transient demand. For reproducibility, the

source code, the driver scripts, the exact JVM flags, and the raw measurements will be made available [repository or archive link].

V. EXPERIMENTAL RESULTS AND ANALYSIS

1. Spatial Memory Scaling

Table V presents the profiling results for the move-recording implementation. The retained heap approximately doubles for every unit increase in N, which doubles the move count, and this agrees with the $\Theta(2^N)$ heap model of Section III. The pure recursive implementation, which retains no history, was measured separately and used only $O(N)$ stack space, below 1 KB for all tested N, in line with the low memory footprint reported for Java by Durrani et al. [9].

Table 5: Java Spatial Memory Profiling Metrics

N	Total moves ($2^N - 1$)	Stack depth	Time (ms)	Heap memory
5	31	5 frames	< 0.1	~2 KB
10	1,023	10 frames	~0.3	~48 KB
15	32,767	15 frames	~4.2	~1.2 MB
18	262,143	18 frames	~28	~9.8 MB
20	1,048,575	20 frames	~112	~38.5 MB
22	4,194,303	22 frames	~456	~154 MB
25	33,554,431	25 frames	~3,720	~1.18 GB

At N = 25 the heap demand reaches 1.18 GB, which exceeds the -Xmx256m configuration and produces an OutOfMemoryError unless the heap is tuned explicitly (the N = 25 runs reported in Table V used -Xmx2g, Section IV-C). The practical limit of the state-recording approach on standard JVM configurations therefore lies at approximately N = 22 to 24, and N = 20 is the recommended ceiling for environments without dedicated memory tuning. At N = 20 the retained heap corresponds to approximately 37 to 39 bytes per move, against the 28 bytes predicted by the model (Section IV-D). The difference is of the order of the reference-array overhead described in Section IV-B, although the present data do not separate the two contributions.

Heap readings at $N = 5$ showed minor variance across trials, which is attributed to JVM warm-up effects [consistent with the ABR concept of [9], [12]: define ABR and confirm these references]. The measured growth ratios approach the theoretical factor only as N increases. Between $N = 5$ and 10 and between $N = 10$ and 15 the observed ratios (about 24 and 25) fall 22 to 25 percent below the theoretical value of 32, because a fixed baseline allocation dominates at small N . From $N = 15$ onwards the ratios (about 8.2 from $N = 15$ to 18, 3.9 from 18 to 20, 4.0 from 20 to 22, and 7.7 from 22 to 25) lie within about 5 percent of the theoretical values of 8, 4, 4, and 8. $N = 15$ therefore marks the point beyond which the spatial measurements follow the asymptotic model in this configuration.

2. Cross-Language Execution Time Context

Table VI reproduces selected execution times from Durrani et al. [9] to place the spatial findings within the cross-language picture. The shorter Java time relative to C++ in that study is consistent with the $O(N)$, stack-only profile of the pure recursive regime confirmed in Table V: the move-printing implementation allocates almost no heap, so the collector has little work to do. This is an interpretation, because the present experiments measured memory and did not isolate its effect on execution time.

TABLE 6: Cross-Language Toh Execution Times
(From [9])

N	Python (s)	C++ (s)	Java (s)
5	0.000004	0.000001	0.0000125
15	0.0216	0.00145	0.000616
20	0.179	0.01808	0.005754
25	5.430	0.352	0.08935
30	173.16	11.133	5.405
35	7541.5	305.47	103.92

3. Connecting to Cross-Language Performance

The findings help to explain the Java advantage reported by Durrani et al. [9], who did not measure memory themselves. In the pure recursive, move-printing regime of that study, Java holds only N call frames on the stack and negligible heap, so JIT-compiled code can run with little garbage-collection pressure. At $N = 35$ this coincides with the reported

103.9 s for Java against 305.5 s for C++ (Table VI), although C++ is generally expected to be faster on memory-intensive workloads. The present results indicate that the $O(N)$ footprint is a condition for this outcome and that it does not carry over once history is retained: at $N = 25$ the state-recording variant took about 3.7 s (Table V), compared with 0.089 s for the pure recursive Java version in [9], although the two sets of measurements were obtained on different hardware and are not directly comparable.

4. Recursive vs. State-Recording Comparison

Table VII compares the two implementation strategies across six characteristics. The failure thresholds depend on the configured limits ($-Xss$ for the stack and $-Xmx$ for the heap), so they are indicative rather than absolute; [state the $-Xmx$ value at which the threshold $N > 27$ was determined].

Table 7: Recursive Vs. State-Recording Comparison

Characteristic	Recursive (no history)	State-recording
Space complexity	$O(N)$ stack	$O(2^N)$ heap
Memory region	JVM call stack	JVM heap
GC interaction	Minimal	Heavy (young-generation promotion)
OOM risk	StackOverflowError ($N > 10,000$)	OutOfMemoryError ($N > 27$)
History retained	No	Full sequence
Scalability	Limited by $-Xss$	Limited by $-Xmx$

5. Cross-Environment and Cross-JVM-Version Validation

To address the concern that Table V reflects a single JVM build and machine, the full benchmark of Section IV was independently re-executed on a second, materially different environment: OpenJDK 21.0.10 (HotSpot 64-bit Server VM) on Ubuntu 24.04 LTS, confined to a single CPU core and 3.9 GB of RAM, a configuration close to a typical containerised production deployment (Section I). The benchmark was run in full twice with the same $-Xmx2g -XX:+UseSerialGC$ configuration and five-trial median protocol as in Section IV, and Table VIII reports the mean of the two per- N medians.

Table 8: Cross-Jvm-Version Validation (Openjdk 21.0.10, Ubuntu 24.04, Single-Core Container)

N	Total moves ($2^N - 1$)	Time (ms)	Heap memory
5	31	0.04	~336 KB
10	1,023	0.10	~336 KB
15	32,767	0.39	~1.37 MB
18	262,143	2.78	~10.2 MB
20	1,048,575	28.75	~28.7 MB
22	4,194,303	318.2	~146.5 MB
25	33,554,431	4,433	~1.12 GB

The $\Theta(2^N)$ heap-doubling pattern reproduced on this independent environment, and the retained heap at a given N was identical across the two repeat runs to the precision reported, as expected because the allocation pattern is deterministic once N and the JDK build are fixed. Absolute heap figures differ from the OpenJDK 17.0.9 measurements by a factor that varies with N: at N = 18 the new figure is about 4 percent higher (10.2 MB against 9.8 MB), at N = 20 about 25 percent lower (28.7 MB against 38.5 MB), and at N = 22 and N = 25 about 5 percent lower (146.5 MB against 154 MB, and 1.12 GB against 1.18 GB). These differences do not affect the asymptotic claim and are of the kind anticipated in Section IV-

6. The lower value at N = 20 is consistent with differences in backing-array and thread-local allocation buffer (TLAB) overhead between JDK releases, although this explanation was not tested. Execution times differ more widely (28.75 ms against about 112 ms at N = 20, but 4,433 ms against about 3,720 ms at N = 25), reflecting the different hardware, JIT behaviour, and single-core confinement; time is a secondary measure in this study.

Re-running the single-N listing of Listing 3 with N = 25 and default JVM flags on this environment reproduced an OutOfMemoryError, which confirms the heap-exhaustion boundary of Section V-A independently of JVM vendor and version. A further run with the default G1 collector (-XX:+UseG1GC) in place of the Serial collector reproduced the same exponential heap growth, with a smaller idle-heap baseline at low N and a larger retained heap at N =

25 (about 1,231 MB against 1,175 MB), consistent with the region-based and less aggressive reclamation policy of G1. Collector choice therefore shifts the constant factor of the measurement but not its exponential shape, and it should be reported alongside any absolute heap figure in future replications of this benchmark.

VI. JVM MEMORY ARCHITECTURE INSIGHTS

1. Object Header Overhead

Every Java object on the heap carries a header managed by the JVM. On a 64-bit HotSpot JVM with compressed ordinary object pointers (the default for heaps below about 32 GB), the header consists of an 8-byte mark word and a 4-byte compressed class pointer, 12 bytes in total. The three primitive fields of the Move class used in Listing 3 (an int and two chars, $4 + 2 + 2 = 8$ bytes) bring the object to 20 bytes, and 4 bytes of alignment padding round it up to 24 bytes, so the header alone accounts for 50 percent of each object (Table IX). This is the layout assumed by the analytical model of Section III-E.

Table 9: Jvm Object Memory Layout of the Move Class (Compressed Oops)

Field	Type	Size (B)	Notes
Mark word (header)	Internal	8	GC, hash, and lock bits
Class pointer (header)	Internal	4	Compressed
disk	int	4	Primitive
from	char	2	Primitive
to	char	2	Primitive
Alignment padding	—	4	8-byte boundary
Total per Move object	—	24	Header = 50% of total

For N = 20, the 1,048,575 Move objects have a theoretical floor of $1,048,575 \times 24$ bytes, approximately 24 MB. Adding the 4-byte reference to each object gives the 28 bytes per move of Eq. (6), or about 28 MB (Table III). The measured retained heap of about 38.5 MB (Table V) exceeds this prediction by roughly 10 MB. Part of the difference

is the slack in the ArrayList backing array, which grows by a factor of 1.5 and may therefore hold up to $1.5 \times 1,048,575$ reference slots of 4 bytes each, approximately 6 MB (Section IV-B). Accounting for this slack still leaves about 8 MB unexplained, which is plausibly due to thread-local allocation buffers (TLABs) and other residual allocations retained at the time of measurement; this attribution was not isolated in the present experiments, and the pre-sized list variant of Table IV would be the natural means of testing it.

2. Garbage Collection and Generation Promotion

Move objects are first allocated in the young generation (Eden). As Eden fills during execution, minor collections promote the surviving objects, which are those referenced by the ArrayList, to the old generation, so that for large N the old generation holds most of the move history by the time the algorithm completes. Reclaiming it requires a full collection once the list has become unreachable, which is considerably more expensive than a minor collection and may introduce stop-the-world pauses. This sustained old-generation occupancy is consistent with the elevated heap readings observed after completion when no explicit `System.gc()` call is made, and it underscores why forced collection before the baseline reading (Section IV-D) is essential to the accuracy of the measurement.

3. JVM Tuning Recommendations

For applications that must record the full move sequence at $N \geq 20$, four measures follow from the results. First, the maximum heap (`-Xmx`) should be set to at least twice the estimated requirement, which allows for the transient copy made when the backing array grows. Second, the G1 collector (`-XX:+UseG1GC`, the default since JDK 9) is appropriate for large heaps because it avoids long stop-the-world pauses, although Section V-E shows that it retained somewhat more heap than the Serial collector at $N = 25$, so the choice trades footprint against pause time. Third, the list should be pre-sized with `new ArrayList<>(1 << N)`, which removes the resize reallocations and reduces the reference-array overhead from up to 1.5 times to 1.0 times the element count. Fourth, compressed ordinary object pointers should remain enabled (`-XX:`

`+UseCompressedOops`, the default for heaps up to about 32 GB), which saves about 6 MB on the backing array at $N = 20$ relative to 8-byte references.

Limitations and Threats to Validity

The following limitations extend those stated in Section IV-F and bound the generality of the results. First, Section V-E shows that the $\Theta(2^N)$ heap-growth pattern and the heap-exhaustion boundary reproduce across two JVM versions (OpenJDK 17.0.9 and 21.0.10) and two host configurations (a 16 GB workstation and a single-core container with 3.9 GB of RAM), but all measurements remain confined to the HotSpot implementation. Behaviour under OpenJ9, GraalVM, or the Android runtime may differ materially, particularly in object header layout and collector pause characteristics.

Second, the Runtime API reading (`totalMemory() - freeMemory()`) reports used heap, which includes unreachable objects that have not yet been collected, and not the resident set size of the process. Concurrent JIT-compiler and collector threads can introduce small measurement noise even after forced collection, which is why the asymptotic conclusions of this study are drawn from $N \geq 15$ (Section V-A). Third, the workload is a synthetic microbenchmark. Real applications rarely retain a Tower of Hanoi move history in isolation, and interactions with other live objects on a shared heap, such as fragmentation and promotion pressure from unrelated allocations, are not modelled.

Fourth, the collector comparison of Section V-E covers G1 against the Serial baseline only. A systematic comparison that adds ZGC and Shenandoah, identified as future work in Section VIII, would characterise the pause-time trade-offs of the state-recording regime more fully. Finally, the per-object cost model of Section III-E assumes that compressed ordinary object pointers are active. On heaps above about 32 GB, where they are disabled by default, a reference occupies 8 bytes instead of 4, so the cost per recorded move rises from 28 to about 32 bytes, while the Move object itself remains 24 bytes provided that compressed class pointers stay enabled. All reported heap

figures would then shift upward by a constant factor, and the $\Theta(2^N)$ conclusion would be unaffected.

VII. CONCLUSION AND FUTURE WORK

This paper has investigated the spatial behaviour of the Tower of Hanoi algorithm on the Java Virtual Machine, combining an analytical footprint model with a controlled heap-measurement benchmark, and has related the outcome to the cross-language execution-time findings of Durrani et al. [9]. Three principal results emerge. First, the pure recursive implementation needs only $\Theta(N)$ call-stack space, below 1 KB for all tested N , which is the memory condition associated with the Java execution-time advantage over C++ reported in [9]. Second, the state-recording implementation grows as $\Theta(2^N)$ in heap space. Its retained heap approximately doubles with each additional disk once $N \geq 15$, and the roughly 154 MB retained at $N = 22$ implies that a -Xmx256m configuration is exhausted at $N = 23$, a pattern that reproduced on a second JVM version and host configuration (Section V-E). Third, object-level constants matter as much as the asymptotic class: the 12-byte object header accounts for 50 percent of each 24-byte Move object, and the measured heap of 38.5 MB at $N = 20$ exceeds the 24 MB object-size floor by approximately 60 percent, the difference arising from reference-array slack and residual allocations (Section VI-A).

These findings support practical guidance for implementers. Retaining the full move history should be avoided where possible, and moves should instead be generated lazily and streamed to the consumer (Section III-F). Where history is required, $N = 20$ is a prudent ceiling without dedicated memory tuning, and the tuning measures of Section VI-C (adequate -Xmx headroom, pre-sized lists, compressed ordinary object pointers, and a collector chosen with its footprint and pause-time trade-off in mind) extend the usable range. The comparative framework of Table VII allows a practitioner to select between stack-resident and heap-resident strategies on the basis of the failure mode and the limiting JVM parameter that each implies.

Several directions remain for future work: (i) an equivalent spatial analysis of the Python and C++ implementations studied in [9], which would give a unified cross-language comparison of spatial complexity; (ii) an extension of the collector comparison of Section V-E to ZGC and Shenandoah, to characterise the pause-time overhead of the state-recording regime; (iii) re-benchmarking with the value types proposed in Project Valhalla, to quantify the potential saving from eliminating the 12-byte object header; and (iv) a container-aware study under Docker and Kubernetes memory limits, where default JVM heap and stack sizing may conflict with the memory budget imposed by the container.

Acknowledgment

The authors thank the Department of Computer Science and Engineering, Ghousia College of Engineering, Ramanagram, for providing the computational resources and academic environment that made this research possible. The authors also acknowledge the cross-language performance study of Durrani et al. [9], whose observations on Java's spatial efficiency in the Tower of Hanoi problem motivated the detailed spatial memory investigation presented in this paper.

REFERENCES

1. A. Levitin, Introduction to the Design and Analysis of Algorithms, 3rd ed. Boston, MA, USA: Pearson, 2012.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA, USA: MIT Press, 2022.
3. É. Lucas, Récréations Mathématiques, vol. 3. Paris, France: Gauthier-Villars, 1893.
4. "TIOBE Programming Community Index," TIOBE Software BV. Accessed: Oct. 6, 2026. [Online]. Available: <https://www.tiobe.com/tiobe-index/>
5. B. Stroustrup, The C++ Programming Language, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2013.
6. T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, The Java Virtual Machine Specification, Java SE 21 ed. Oracle America, Inc., 2023.
7. "gc — Garbage collector interface," Python 3 Documentation, Python Software Foundation.

- Accessed: Oct. 6, 2026. [Online]. Available: <https://docs.python.org/3/library/gc.html>
8. R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Boca Raton, FL, USA: CRC Press, 2012.
9. [Durrani et al., cross-language performance study of the Tower of Hanoi in Java and C++. Insert the full author list, title, venue, volume, pages, and year from your source.]
10. C. Cubukcu, Z. B. Guven Aydin, and R. Samli, "Comparison of C, Java, Python and Matlab programming languages for Fibonacci and Towers of Hanoi algorithm applications," *Bol. Soc. Parana. Mat.*, vol. 41, 2023, doi: 10.5269/bspm.52209.
11. [Study of sorting-algorithm performance in Java and Python (your earlier reference [10]). Insert full details.]
12. [Study of sorting-algorithm performance in Java and Python (your earlier reference [11]). Insert full details.]
13. "Control Group v2," *The Linux Kernel Documentation*. Accessed: Oct. 5, 2026. [Online]. Available: <https://docs.kernel.org/admin-guide/cgroup-v2.html>
14. "Resource management for pods and containers," *Kubernetes Documentation*. Accessed: Oct. 5, 2026. [Online]. Available: <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>
15. R. W. Sebesta, *Concepts of Programming Languages*, 11th ed. Boston, MA, USA: Pearson, 2016.
16. P. Naur and B. Randell, Eds., *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, Garmisch, Germany, Oct. 7–11, 1968. Brussels, Belgium: NATO Science Affairs Division, 1969.
17. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1978.
18. D. M. Ritchie, "The development of the C language," in *Proc. 2nd ACM SIGPLAN Conf. History of Programming Languages (HOPL-II)*, Cambridge, MA, USA, 1993, pp. 201–208.
19. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Reading, MA, USA: Addison-Wesley, 1983.
20. C. A. R. Hoare, "Hints on programming language design," *Stanford Univ.*, Stanford, CA, USA, Tech. Rep. STAN-CS-73-403, Dec. 1973.
21. J. Gosling et al., *The Java Language Specification*, Java SE 21 ed. Oracle America, Inc., 2023.
22. "History and license," *Python 3 Documentation*, Python Software Foundation. Accessed: Oct. 6, 2026. [Online]. Available: <https://docs.python.org/3/license.html>
23. "sys — System-specific parameters and functions," *Python 3 Documentation*, Python Software Foundation. Accessed: Oct. 6, 2026. [Online]. Available: <https://docs.python.org/3/library/sys.html>
24. A. M. Hinz, S. Klavžar, U. Milutinović, and C. Petr, *The Tower of Hanoi—Myths and Maths*, 2nd ed. Cham, Switzerland: Birkhäuser/Springer, 2018.
25. P. Buneman and L. Levy, "The Towers of Hanoi problem," *Inf. Process. Lett.*, vol. 10, no. 4–5, pp. 243–244, 1980.
26. M. C. Er, "An iterative solution to the generalized Towers of Hanoi problem," *BIT*, vol. 23, pp. 295–302, 1983, doi: 10.1007/BF01934458.
27. J. S. Wu and R. J. Chen, "The Towers of Hanoi problem with parallel moves," *Inf. Process. Lett.*, vol. 44, no. 5, pp. 241–243, 1992, doi: 10.1016/0020-0190(92)90207-C.
28. "Java Object Layout (JOL)," *OpenJDK Code Tools Project*. Accessed: Oct. 5, 2026. [Online]. Available: <https://openjdk.org/projects/code-tools/jol/>
29. M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," in *Proc. 20th ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, San Diego, CA, USA, 2005, pp. 313–326.
30. A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in *Proc. 22nd ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, Montreal, QC, Canada, 2007, pp. 57–76.
31. S. M. Blackburn et al., "The DaCapo benchmarks: Java benchmarking development and analysis,"

- in Proc. 21st ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA), Portland, OR, USA, 2006, pp. 169–190.
32. L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, Oct. 2000, doi: 10.1109/2.876288.
 33. R. Pereira et al., "Energy efficiency across programming languages: How do energy, time, and memory relate?," in Proc. 10th ACM SIGPLAN Int. Conf. Softw. Lang. Eng. (SLE), Vancouver, BC, Canada, 2017, doi: 10.1145/3136014.3136031.