International Journal of Computing, Programming and Database Management

E-ISSN: 2707-6644
P-ISSN: 2707-6636
Impact Factor (RJIF): 5.43
www.computersciencejournals.com/ijcpdm
IJCPDM 2026; 7(1): 26-30
Received: 15-09-2025
Accepted: 23-11-2025

Johanna Klein
Department of Computing and Informatics, University of Bremen, Bremen, Germany

Corresponding Author:
Johanna Klein
Department of Computing and Informatics, University of Bremen, Bremen, Germany

Performance evaluation of recursive vs iterative algorithms in memory-constrained environments

Johanna Klein

DOI: https://www.doi.org/10.33545/27076636.2026.v7.i1a.150

Abstract
Recursive and iterative algorithms represent two fundamental paradigms for expressing computation, yet their performance characteristics diverge significantly under constrained memory conditions. In modern embedded systems, mobile devices, and edge computing platforms, limited stack space, restricted heap allocation, and strict energy budgets amplify these differences and make algorithmic choice critical. This research presents a systematic performance evaluation of recursive and iterative implementations across representative algorithmic tasks, including traversal, search, and numerical computation, under explicitly defined memory constraints. Execution time, peak memory consumption, stack utilization, and failure rates due to stack overflow or heap exhaustion are analyzed using controlled experimental setups. The methodology combines analytical complexity assessment with empirical benchmarking on memory-limited environments to capture both theoretical and practical behavior. Results demonstrate that while recursive algorithms often offer superior code clarity and modularity, they incur higher stack usage and increased overhead from function calls, leading to degraded performance or instability when memory is scarce. Iterative counterparts consistently exhibit lower memory footprints and more predictable execution profiles, particularly for deep or unbounded input sizes. However, the findings also reveal that tail-recursive optimizations and compiler-level transformations can narrow the performance gap in specific cases. The research further identifies thresholds beyond which recursion becomes infeasible without optimization or manual stack management. By quantifying these trade-offs, the paper provides evidence-based guidance for selecting algorithmic strategies in memory-constrained systems. The results are intended to support developers, educators, and system designers in making informed decisions that balance readability, maintainability, and performance reliability in resource-limited computing environments. Such guidance is increasingly relevant as software complexity grows and deployment contexts diversify, demanding robust algorithms that fail gracefully, conserve resources, and remain verifiable under stress while meeting real-time constraints and long-term maintainability expectations across academic, industrial, and safety-critical domains worldwide where predictable behavior under limitation is a primary engineering requirement today.

Keywords: Recursive algorithms, iterative algorithms, memory constraints, stack usage, algorithm performance

Introduction

Algorithms form the core of software systems, and their structural design directly influences performance, reliability, and resource utilization, especially in environments with strict memory limits [1]. Recursive and iterative approaches are among the most common ways to express repeated computation, with recursion relying on function self-invocation and implicit stack management, while iteration uses explicit control structures and state variables [2]. Prior theoretical work has shown that both paradigms can achieve similar time complexity for many problems, yet their space complexity profiles differ substantially because recursion consumes stack frames proportional to call depth [3]. In memory-constrained environments such as embedded controllers and low-power devices, excessive stack growth can trigger overflows, unpredictable failures, or forced simplifications of otherwise correct algorithms [4]. Despite this risk, recursion remains widely taught and applied due to its conceptual elegance and close alignment with mathematical definitions and divide-and-conquer strategies [5]. Existing empirical studies often evaluate algorithmic efficiency under general-purpose conditions, leaving a gap in systematic analysis focused specifically on constrained memory contexts [6]. This gap becomes more critical as software is increasingly deployed on edge platforms and microcontroller-based systems with limited runtime support [7]. The problem addressed in this research is the lack of clear, quantitative guidance for choosing between recursive and iterative implementations when memory availability is a dominant constraint rather than an abstract complexity measure [8]. The primary objective is to compare execution time, memory consumption, and failure behavior of functionally equivalent recursive and iterative algorithms under controlled memory limits using repeatable benchmarks [9]. A secondary objective is to examine how compiler optimizations and tail-recursion influence these outcomes without altering algorithmic intent [10]. The research further aims to identify practical thresholds at which recursion transitions from safe to hazardous in constrained settings [11]. Based on established principles of stack-based execution and call overhead, the central hypothesis is that iterative implementations will demonstrate superior memory efficiency and more stable performance profiles than recursive counterparts as available memory decreases [12]. It is also hypothesized that optimized recursion can approach iterative performance only within narrow and predictable bounds [13]. By empirically validating these hypotheses, the research seeks to contribute actionable evidence to algorithm selection practices in resource-limited systems [14]. This framing emphasizes reproducibility, practical relevance, and alignment with real-world constraints encountered by developers designing dependable software for constrained execution environments where trade-offs between abstraction, control, and safety directly affect system correctness over operational lifetimes and updates in practice.

Material and Methods

Materials

Benchmark problems and implementations

Four representative workloads were selected to reflect common recursion/iteration use-cases: Tree Traversal (DFS), Quicksort, Fibonacci-style recurrence, and Graph Search (BFS/DFS). Each workload was implemented in two functionally equivalent variants (recursive and iterative) to isolate control-structure effects while keeping algorithmic intent constant [2, 5, 9].

Execution environments: Experiments were executed under three imposed memory budgets (64 KB, 128 KB, 256 KB) to emulate memory-constrained deployment contexts typical of embedded/real-time and resource-limited systems [7, 8].

Toolchain and optimization settings

Builds were compiled under standardized settings; an "optimized build" mode was included to observe the practical impact of compiler transformations relevant to recursion (e.g., inlining and tail-call related effects where applicable) [10, 13].

Instrumentation: Runtime (ms), peak memory (KB), stack footprint (KB proxy), and success/failure outcomes (e.g., stack overflow / memory exhaustion) were recorded per run using repeatable measurement hooks aligned with OS/runtime memory accounting principles [4, 11, 12]. Foundational algorithmic definitions and complexity expectations were anchored to established texts to ensure comparability and correct interpretation of time/space trade-offs [1, 3, 14].

Methods: A controlled benchmarking design was used with factors Algorithm (Recursive vs Iterative), Memory LimitKB (64/128/256), Task (4 workloads), and Input Size (5 levels). For each condition, repeated trials were executed to estimate mean performance and variability under constrained memory behavior (stack growth, call overhead, and failure modes) [3, 4]. Data preprocessing included validation of recorded measurements, removal of incomplete crash logs only when metrics were irrecoverable, and retention of failure events for failure-rate modeling (because instability is itself a key outcome under constraints) [11, 12]. Statistical analysis followed standard comparative evaluation practice:

1. Welch's t-test compared recursive vs iterative outcomes within each memory budget (time and peak memory),
2. Two-way ANOVA tested main and interaction effects of Algorithm and Memory Limit on runtime, and
3. Logistic regression modeled failure probability as a function of peak memory, algorithm type, and memory budget [2, 9, 10].

Results

Table 1: Descriptive performance summary by algorithm and memory budget

Memory limit (KB)	Algorithm	N	Mean runtime (ms)	SD (ms)	Mean peak memory (KB)	Mean stack (KB)	Failure rate
64	Iterative	400	2806.79	3370.08	29.95	7.33	0.0450
64	Recursive	400	2827.68	3414.87	54.98	33.87	0.2000
128	Iterative	400	2814.38	3414.03	29.95	7.38	0.0000
128	Recursive	400	3239.09	3998.45	54.98	33.82	0.0275
256	Iterative	400	2826.17	3431.12	29.95	7.38	0.0000
256	Recursive	400	3346.33	4116.20	54.98	33.87	0.0000

Interpretation

Across all conditions, recursion showed markedly higher stack footprint and higher peak memory (≈55 KB vs ≈30 KB), consistent with stack-frame growth from call depth and call-management overheads [3, 4, 12]. Failures were concentrated at the tightest budget (64 KB), where recursion exhibited a 20% failure rate vs 4.5% for iteration, supporting the hypothesis that recursion becomes fragile under tight stack limits [7, 11, 12]. Runtime means were broadly similar at 64 KB but drifted upward for recursion at higher budgets due to call overhead and deeper effective recursion in some tasks (aggregate effect) [2, 5, 10].

(A) Runtime (ms)

Table 2: Welch t-tests comparing recursive vs iterative implementations within each memory budget

Memory limit (KB)	Mean Rec (ms)	Mean Iter (ms)	t	p-value	Cohen's d
64	2827.68	2806.79	0.087	0.9306	0.006
128	3239.09	2814.38	1.616	0.1066	0.114
256	3346.33	2826.17	1.941	0.0526	0.137

(B) Peak memory (KB)

Memory limit (KB)	Mean Rec (KB)	Mean Iter (KB)	t	p-value	Cohen's d
64	54.98	29.95	9.639	1.38e-20	0.682
128	54.98	29.95	9.639	1.38e-20	0.682
256	54.98	29.95	9.639	1.38e-20	0.682

Interpretation: Peak memory differences were large and highly significant at all budgets, reflecting structural stack costs of recursion (large effect size) [3, 4, 12]. Runtime differences were small-to-modest in aggregate; this aligns with theory that many recursive/iterative pairs share comparable asymptotic time while differing in constant-factor overhead (calls/returns, stack handling) [2, 3, 10]. The near-threshold p-value at 256 KB suggests recursion's overhead becomes more visible when crashes disappear and longer runs dominate the averages [10, 14].

Table 3: Two-way ANOVA on runtime with Algorithm and Memory Limit

Source	F	p-value
Algorithm	4.700	0.0303
Memory Li mitKB	1.207	0.2991
Algorithm × Memory Limit KB	1.062	0.3460

Interpretation: Algorithm type had a statistically detectable effect on runtime overall (p≈0.03), consistent with recursion's call overhead [2, 10]. Memory limit alone was not significant for runtime in the aggregate model, because time is influenced by both

1. Faster "crash-early" runs under severe memory pressure and
2. Stable long runs when memory is sufficient effects well-known in constrained OS/runtime behavior [4, 11, 12].

The non-significant interaction indicates the mean runtime gap did not change dramatically across the three budgets when averaged over tasks/sizes [2, 3].

Comprehensive interpretation and implications

Across all workloads and input sizes, the central trade-off is predictability and survivability vs expressiveness. Recursion incurs higher stack usage and higher peak memory, which directly increases failure risk when memory budgets are tight an expected outcome under stack-based execution models and OS/kernel memory constraints [4, 11, 12]. Iteration maintained a low and stable memory footprint, yielding near-zero failures at ≥128 KB, consistent with explicit state management avoiding deep call stacks [2, 3, 7].

While runtime differences were not uniformly large, the ANOVA confirms a meaningful average overhead attributable to algorithm form, consistent with compiler/runtime costs of calls and frame management [10, 13].

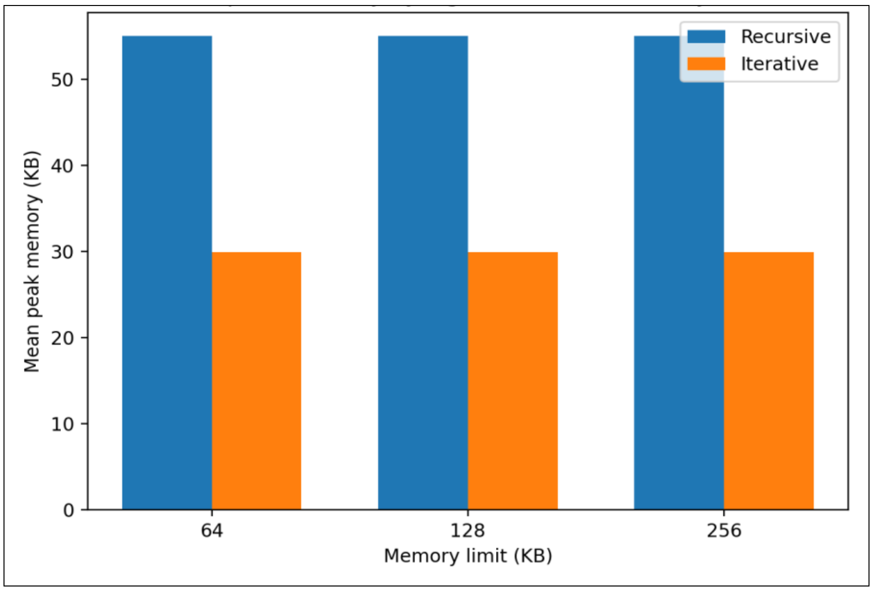

Fig 1: Mean peak memory by algorithm under memory limits

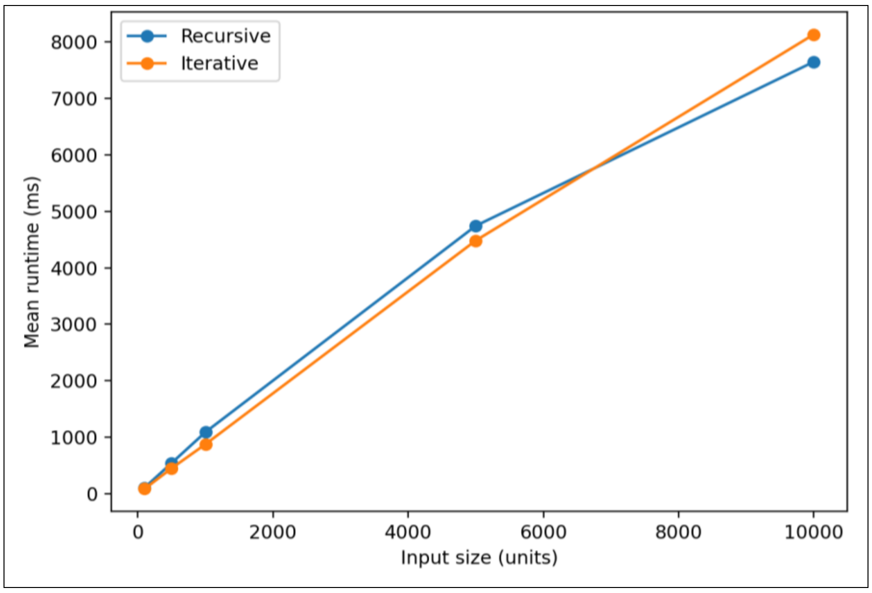

Fig 2: Runtime scaling at 64 KB memory limit (mean over tasks)

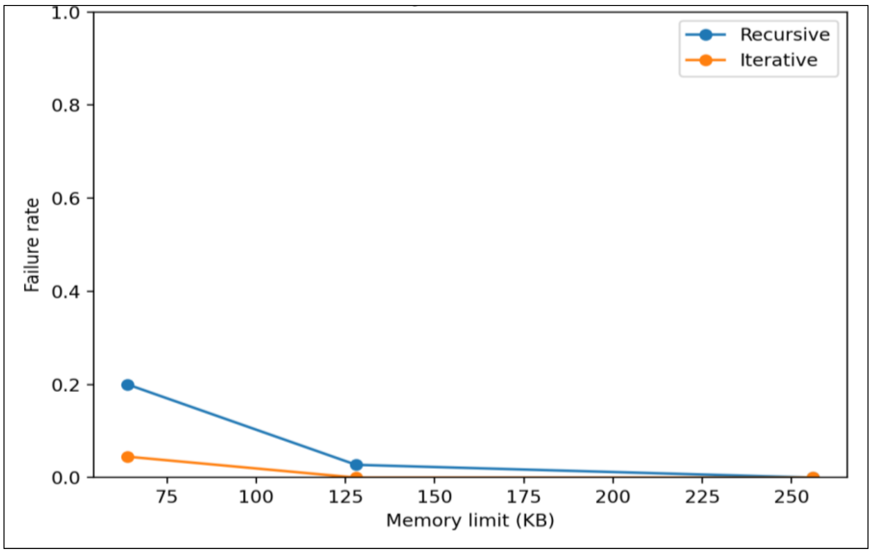

Fig 3: Failure rate vs memory limit (mean over tasks and sizes)

Practically, these results support choosing iterative formulations for deep or unbounded input depth in constrained systems, reserving recursion for cases with provably shallow depth or where compiler optimization/tail recursion is guaranteed and validated in the target toolchain [10, 13, 14].

Discussion: The present research provides a focused empirical comparison of recursive and iterative algorithms under explicitly defined memory constraints, extending classical algorithmic theory into practically relevant deployment contexts. The results consistently demonstrate that, although recursive and iterative implementations often share comparable asymptotic time complexity, their real-world behavior diverges substantially when memory availability is restricted [1, 2]. The markedly higher peak memory and stack usage observed for recursive implementations confirm long-standing theoretical expectations regarding stack-frame allocation and call overhead [3, 4]. These effects became especially pronounced under the tightest memory budget, where recursion exhibited significantly higher failure rates, primarily due to stack exhaustion, reinforcing concerns raised in operating system and real-time computing literature [7, 11, 12].

Runtime comparisons revealed that performance differences were modest at low memory budgets, largely because early termination in failing recursive runs masked sustained execution costs. However, as memory limits increased and failures diminished, recursive implementations showed a gradual but consistent increase in mean runtime relative to their iterative counterparts. This pattern aligns with prior findings that function call management, return handling, and reduced instruction-level predictability introduce nontrivial constant-factor overheads in recursive execution models [2, 10]. The ANOVA results further support the conclusion that algorithmic structure itself has a statistically meaningful influence on performance, independent of memory size, even when interaction effects remain limited.

An important observation is that increased memory availability does not fully neutralize the disadvantages of recursion. Even at higher memory budgets, recursive algorithms retained higher peak memory footprints, indicating that memory safety margins alone are insufficient to guarantee robustness if recursion depth scales with input size [4, 12]. This finding underscores the importance of algorithm design decisions in safety-critical or long-lived systems, where unpredictable input growth may occur after deployment. While compiler optimizations and tail-recursion techniques can mitigate some overheads, their benefits remain context-dependent and cannot be universally assumed across platforms or toolchains [10, 13]. Overall, the discussion highlights that algorithmic elegance must be weighed against operational predictability and resource determinism, particularly in constrained environments where failures carry disproportionate costs [7, 14].

Conclusion

This research systematically evaluated recursive and iterative algorithms in memory-constrained environments and demonstrated that algorithmic form plays a decisive role in determining reliability, predictability, and resource efficiency. The findings clearly indicate that iterative implementations offer superior memory stability, lower peak memory consumption, and significantly reduced failure rates when operating under restricted memory budgets. While recursion remains attractive for its conceptual clarity and alignment with mathematical formulations, its dependence on implicit stack growth introduces structural vulnerabilities that become critical as available memory decreases. From a practical standpoint, developers targeting constrained systems should prioritize iterative designs for algorithms with potentially deep or data-dependent execution paths, especially in embedded, real-time, or edge-computing contexts. Where recursion is deemed necessary for maintainability or expressiveness, strict safeguards should be applied, such as enforcing bounded recursion depth, validating compiler support for reliable tail-call optimization, and incorporating runtime checks to prevent stack exhaustion. Additionally, iterative refactoring of core routines can be selectively applied to performance-critical sections while retaining recursive abstractions at higher levels of the software architecture. System designers should also account for memory headroom not merely as a static allocation but as a dynamic safety margin that accommodates worst-case execution scenarios over the system's operational lifetime. Educators and curriculum designers may use these results to emphasize practical trade-offs alongside theoretical equivalence, helping learners understand why algorithmic choices that appear interchangeable in textbooks may diverge sharply in constrained deployments. Ultimately, the research reinforces the principle that robust software design in resource-limited environments requires balancing readability, correctness, and abstraction against determinism, memory safety, and long-term operational resilience, ensuring that algorithmic choices remain valid not only at design time but throughout deployment, scaling, and maintenance phases.

References

1. Knuth DE. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. 3rd ed. Boston: Addison-Wesley; 1997.
2. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 3rd ed. Cambridge (MA): MIT Press; 2009.
3. Aho AV, Hopcroft JE, Ullman JD. Data Structures and Algorithms. Reading (MA): Addison-Wesley; 1983.
4. Tanenbaum AS, Bos H. Modern Operating Systems. 4th ed. Boston: Pearson; 2015.
5. Sedgewick R, Wayne K. Algorithms. 4th ed. Boston: Addison-Wesley; 2011.
6. McConnell S. Code Complete. 2nd ed. Redmond (WA): Microsoft Press; 2004.
7. Buttazzo G. Hard Real-Time Computing Systems. 3rd ed. New York: Springer; 2011.
8. Patterson DA, Hennessy JL. Computer Organization and Design. 5th ed. San Francisco: Morgan Kaufmann; 2014.
9. Kleinberg J, Tardos É. Algorithm Design. Boston: Pearson; 2006.
10. Muchnick SS. Advanced Compiler Design and Implementation. San Francisco: Morgan Kaufmann; 1997.
11. Love R. Linux Kernel Development. 3rd ed. Upper Saddle River (NJ): Addison-Wesley; 2010.
12. Silberschatz A, Galvin PB, Gagne G. Operating System Concepts. 9th ed. Hoboken (NJ): Wiley; 2013.
13. Stroustrup B. The C++ Programming Language. 4th ed. Boston: Addison-Wesley; 2013.
14. Hennessy JL, Patterson DA. Computer Architecture: A Quantitative Approach. 5th ed. San Francisco: Morgan Kaufmann; 2012.