International Journal of Innovation and Applied Studies
ISSN 2028-9324 Vol. 47 No. 2 Dec. 2025, pp. 250-255

http://www.ijias.issr-journals.org/

Comparative study of the temporal and special algorithmic complexity of iterative structures versus recursive structures

DIONGA NDIBU Ornella, LUBONGO MUEMBE Georgine, Glory ALONDA MADOMBA, Simplice EALE BOTULI, and KABEYA TSHISEBA Cedric

Département de Mathématique et Informatique, Faculté de Sciences, Université Pédagogique Nationale (UPN), Ngaliema, Kinshasa, RD Congo

ABSTRACT: This article focuses on the comparison of the temporal and spatial algorithmic complexities of iterative and recursive structures, through the analysis of classic cases (*factorial, Fibonacci sequence, tree traversal)*.
We demonstrate in this study that on the one hand iteration generally offers better memory efficiency (*O (1) in many cases)* and avoids the risks of stack overflow, *and* on the other hand that recursion, although more elegant and intuitive for certain problems (such as tree traversals), can generate a memory overload (*O (n) in call stack)* and degraded time complexity in non-optimized cases (eg: naive Fibonacci in $O(2^n)$).
The results obtained here highlight that the choice between these two approaches depends on the context the developer is in. It is therefore worth noting that iteration is better suited to linear and memory-constrained problems, and recursion to nested structures (*trees, divide-and-conquer)*, especially if the language supports tail call optimization.
This comparison provides objective criteria to guide developers in selecting the most effective approach based on needs.

KEYWORDS: iteration, recursion, time complexity, space complexity, call stack, optimization.

1 INTRODUCTION

In algorithms, problem solving frequently depends on dreary structures, actualized basically agreeing to two ideal models: **iteration** (*for for and while loops)* and **recursion** (*for self-calling functions)*. Whereas both approaches accomplish the same comes about, their suggestions in terms of execution, meaningfulness, and proficiency contrast altogether.

Algorithm optimization may be a central issue in computer science, especially in requesting areas such as high-performance computing, counterfeit insights, and implanted frameworks.

It ought to be famous that the choice between cycle and recursion specifically impacts **the time complexity,** i.e. the execution time of a code, **the space complexity,** i.e. the memory utilized, and the viability of the code.

In spite of the fact that a recursive usage is frequently more natural for issues like tree traversal or divide-and-conquer calculations (*e.g., consolidate sort)*, it can bring about memory overhead due to the call stack. Alternately, iterative arrangements, whereas in some cases less rich, by and large offer way better memory effectiveness.

Through this study, we will try to answer the following two questions:

1. In which cases is iteration more efficient than recursion (*and vice versa)* ?
2. How does temporal and spatial complexity evolve depending on the chosen paradigm?

For clarity, we will answer our questions posed above while relying on three classic algorithmic problems below:

1. Calculation of the factorial,
2. Fibonacci sequence,
3. Binary tree traversal.

For each case, we will systematically compare on the one hand the iterative, recursive implementations and their time complexity, and on the other hand their space complexity (stack/ *heap memory)* and their robustness against borderline cases.

2 THEORETICAL REMINDERS ON ALGORITHMIC COMPLEXITIES

2.1 ALGORITHM

Here we can define an algorithm as a finite and unambiguous sequence of operations or instructions that solve a problem or obtain a result. With sequences of instructions, variables, tests and loops, we can write all the algorithms in the world.

2.2 ALGORITHMIC COMPLEXITY

In computer science, the complexity of an algorithm corresponds to the **amount of resources required for its execution.** She article author of two following things:

In computer science, the complexity of an algorithm refers to the amount of resources required for its execution. It mainly concerns two aspects:

- **Time complexity**, which measures the execution time of an algorithm;
- **Space complexity**, which measures the additional memory used by an algorithm, beyond the memory required to store the input data.

These complexities are **independent.** For the same algorithm, it is possible to have excellent time complexity while having disastrous space complexity, or vice versa.

Also, it is important to consider how an algorithm behaves in **different usage contexts.** This means testing its performance with a variety of inputs to see how it reacts in the best case, worst case, and average case.

Generally, attention is paid to the **worst-case complexity** in order to already prepare the algorithm for the most difficult situations, and to the **average complexity,** because it represents the usual behavior of the algorithm.

If multiple algorithms exist to solve the same problem, calculating the complexity of each algorithm provides a way to compare their performance.

2.3 CALCULATING COMPLEXITY

To calculate the complexity of an algorithm, whether temporal or spatial, we use the **Big *O* notation,** of the form $O(f(n))$. It allows us to express the complexity of an algorithm as a function of the size of the input, generally denoted *n.* The $f(n)$ in $O(f(n))$ thus represents a **mathematical function** that describes how the complexity of the algorithm increases with *n.*

When using Big O notation, the emphasis is on the behavior of algorithms as the input size becomes very large, thus for large values of *n.*

This strategy disposes of less imperative components, such as constants and minor terms, centering on the term that has the most prominent affect on the development of the work. For illustration, in case the precise complexity of an calculation is $O(3n^2+5n+7)$, we disentangle this to $O(n^2)$, since for huge values of *n,* n^2 is the prevailing term.

Time complexity is evaluated based on the total number of fundamental operations performed in the algorithm. A fundamental operation is an operation performed on a unit of data in constant time. Examples of fundamental operations include comparisons, assignments, array accesses, and also operations that analyze the control structure, such as loops and conditions.

Space complexity is an estimate of the total amount of memory required to run the algorithm, for example, memory for variables or data structures. This often includes stack space for function calls.

2.4 COMMON COMPLEXITY CLASSES

In this part of our study, we will present and classify algorithms based on their efficiency. Knowing and understanding complexity classes is important for choosing the right algorithm to solve the problem you are facing.

Below are some of the most common complexities:

- **Constant complexity *O* (1):** This class means that the algorithm performs a fixed number of operations, or runs in constant time, regardless of the size of the input. A common example is accessing an element of an array via its index.

- **Logarithmic complexity *O* (log (*n)*):** Algorithms with logarithmic complexity decrease the problem space with each iteration. For example, the binary search algorithm halves the search space with each step.
- **Linear complexity *O* (*n)*:** This category indicates that the number of operations performed or the execution time grows proportionally to the size of the input data. Linear search is a classic example, requiring in the worst case to scan the entire list to locate an element.
- **Quasi-linear complexity *O* (*n* · log (*n)*):** In this class, we find several well-known sorting algorithms, such as quick sort and merge sort. In general, their performance is superior to that of insertion sort.
- **Quadratic complexity *O* (n^2):** As noted earlier, quadratic complexity appears in algorithms that have one loop nested within another, each loop looping through the entire data set. Quadratic complexity is a category of polynomial complexity, expressed as *O* (n^2) where *a* represents an integer.
- **Exponential complexity *O* (*2n)*:** The additional elements in the input data here cause each of them to double the number of operations and/or the execution time. Generally speaking, a complexity *O* (*a n)* implies that each new element increases the operations or the execution time by a factor of *a.* A classic example of an algorithm with exponential complexity is the naive implementation of the function that generates a Fibonacci number.
- **Factorial complexity *O* (*n* !):** With algorithms with factorial complexity, each new element increases the total number of operations by multiplying it by the number of existing elements. A classic example is organizing various cities in a travel itinerary, examining all possible routes.

3 COMPARATIVE STUDY ON SELECTED CASES

This section presents a systematic evaluation of iterative and recursive approaches across three well-known algorithmic problems. For each case, we will analyze the implementations, measure the theoretical complexities, and discuss the practical implications.

It is important to point out here that for each case, we will have to present its implementation in Python, all using an iterative structure, then a recursive structure, and finally analyze for these two implementations, depending on the input data, the time complexity on one side and the space complexity on the other.

3.1 CALCULATION OF THE FACTORIAL

- **Problem**

  Calculate n! = n × (n-1) ×... × 1

- **Implementations in Python**

Iterative

```
def fact_iter (n):
res = 1
for i in range (1, n+1):
res *= i return res
```

Recursive

- **Analysis**

```
def fact_rec (n):
if n == 0:
return 1
return n * fact_rec (n-1)
```

Temporal

- Iterative: *O (n)* (*n* multiplications)
- Recursive: *O (n)* (*n* calls)

Spatial

- Iterative: *O (1)* (constant variables)
- Recursive: *O (n)* (call stack)

- **Observation**

We can say from the above implementation that the iterative version is memory- optimal. Recursion, while elegant, actually consumes more space resources.

3.2 FIBONACCI SEQUENCE

- **Issue**

Calculate F (n) = F (n-1) + F (n-2) with F (0) = 0, F (1) = 1

- **Implementations in Python**

Iterative

```
def fib_iter (n):
a, b = 0, 1
for _ in range (n): a, b = b, a + b return a
```

Naive Recursive

- **Analysis**

```
def fib_rec (n):
if n <= 1:
return n
return fib_rec (n-1) + fib_rec (n-2)
```

Temporal

- Iterative: *O (n)*
- recursive: *O (2 n)* (recursion tree)

Spatial

- Iterative: *O (1)*
- Recursive: *O (n)* (stack height)

- **Observation**

We can therefore see from the illustration above the obvious case where iteration radically outperforms naive recursion. The recursive version quickly becomes impractical for, say, *n>30.*

3.3 BINARY TREE TRAVERSAL

- **Problem**

Traverse all nodes of a binary tree in infix order.

- **Python implementations**

Iterative (with explicit stack)

```
def inorder_iter (root):
stack = []
res = []
curr = root
while curr or stack:
while curr:
stack.append (curr)
```

curr = curr.left curr = stack.pop () res.append (curr.val) curr = curr.right

return res

Recursive

- **Analysis**

def inorder_ rec (node, res):

if not node:

return

inorder_rec (node.left, res) res.append (node.val) inorder_rec (node.right, res)

Temporal

- Both: *O (n)* (visit each node once)

Spatial

- Iterative: *O (h)* (*h* = height of the tree)
- Recursive: *O (h)* (call stack)

- **Special cases**

Balanced tree: *h = log (n) → O (log n)* space tree: *h = n → O (n)* space

- **Observation**

In this problem, we can see that both approaches have similar performance. Recursion provides more concise and readable code, while iteration avoids the risk of stack overflow.

4 Discussion Of Results And Recommendations

4.1 Summary Of Key Observations

After all these previous Python implementations for illustration purposes, our comparative analysis reveals several major trends, which we will discuss in terms of memory efficiency, time performance, and maintainability.

1. Memory efficiency:

- Iterative implementations consistently dominate in space complexity

(*O (1) vs O (n) for most recursive cases)*

- Recursion generates a memory overhead proportional to the depth of calls

2. Time performance:

- In simple cases (*factorial)*, the two approaches are equivalent in time (*O (n)*),
- For substructure problems (*Fibonacci)*, naive recursion shows glaring limits (*O (2ⁿ)* vs *O (n)* in iterative)

3. Maintainability:

- Recursion provides superior expressiveness for naturally recursive problems (*trees, divide-and-conquer)*
- Iteration often produces more verbose but more transparent code for sequential operations

4.2 Key Decision Factors

For better decision making by the developer, the choice between iteration and recursion should consider as criteria the *memory constraint,* the *execution depth* and the *sequential problem for* **iterative** structures, and, the *tree/graph, code readability/elegance* and *optimized language for* **recursive** structures.

5 Conclusion

Identifying the complexity of algorithms is fundamental knowledge for every developer, as it allows them to make wise choices during the development process.

This study allowed us, through the three cases discussed above, to make a comparative analysis of the temporal and spatial algorithmic complexity of iterative structures and those of recursive structures, with the aim of helping developers who will be confronted with this type of situation, to be able to easily make their choices, opting either for recursion or for iteration.

Following the above, we therefore confirm the following:

1. Iterative structures are generally more memory efficient, especially for linear problems.
2. Recursion is elegant for nested structures (trees, graphs), but can lead to memory overhead.
3. The choice depends on the problem and the language: Languages that optimize tail recursion (e.g., Scala) reduce the performance gap.

As an example, we can cite future work exploring the impact of compiler optimizations on recursion in different languages.

REFERENCES

[1] Cormen, TH, Leiserson, CE, Rivest, RL, & Stein, C. (2022).
[2] Introduction to Algorithms (4th ed.). MIT Press. [Reference work on algorithmic analysis, covering iterative and recursive complexities].
[3] Sedgewick, R., & Wayne, K. (2019). *Algorithms* (4th ed.). Addison- Wesley.
[4] Comparative analysis of recursive and iterative implementations with empirical benchmarks].
[5] Knuth, DE (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley. [Mathematical modeling of the memory costs of recursive calls].
[6] Harper, R. (2016). Programming in Standard ML. Carnegie Mellon University. [Optimizing recursive calls in functional languages].
[7] Martinez, P., & Roura, S. (2021). « Recursive vs. Iterative Algorithms: A Performance Benchmark». *Journal of Computer Science,* 17 (3), 45-62. *[Experimental comparison on 10 programming languages].*
[8] Kleinberg, J., & Tardos, É. (2005). *Algorithm Design.* Pearson.
[9] Advanced techniques for optimizing recursive solutions].
[10] Aho, AV, Lam, MS, Sethi, R., & Ullman, J.D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Pearson. [Optimizations automatic calls recursive by compilers].
[11] Bryant, RE, & O'Hallaron, DR (2015). Computer Systems: A Programmer's Perspective (3rd ed.). Pearson. [Memory Management and Stack Overflow Risks].
[12] Odersky, M., Spoon, L., & Venners, B. (2021). Programming in Scala (5th ed.). Artima. [Implementing tail recursion in languages modern].
[13] Chen, L., & Patel, Y. (2023). «Algorithmic Paradigms: 40 Years of Iteration vs. Recursion». *ACM Computing Surveys,* 55 (2), 1-38. *[Historical meta-analysis of preferences according to application domains].*