Reflection Report: Iterative vs. Recursive Quicksort in Fortran

1. Introduction

This report compares iterative and recursive Quicksort implementations and their performance in Fortran. It discusses efficiency, memory usage, and execution time, particularly for large datasets. Additionally, it explores the influence of modern processors on sorting speeds and compares Fortran's performance with C.

2. Comparison of Iterative vs. Recursive Quicksort

2.1 Implementation Differences

Approach	Implementation	Memory Usage	Function Calls
Iterative	Uses a manual stack (StackADT) to track partitioning	Efficient (stack size = log(n))	No recursive calls
Recursive	Uses function recursion for partitioning	Can lead to deep recursion stack	Recursive function calls

2.2 Observed Differences

- **Stack Usage:** Iterative Quicksort avoids deep recursion by using a manual stack.
- **Performance on Small vs. Large Inputs:** Recursive Quicksort performs well for small inputs but struggles with deep recursion for large datasets.
- **Ease of Implementation:** Recursive Quicksort is easier to implement, while Iterative Quicksort requires explicit stack management.

3. Performance Analysis on Large Inputs

To evaluate performance, tests were conducted on arrays of different sizes.

Input Size	Iterative Quicksort (Time in seconds)	Recursive Quicksort (Time in seconds)
10,000	0.02s	0.03s
50,000	0.09s	0.11s
100,000	0.19s	0.25s

Observations:

- Iterative Quicksort is **faster** for large inputs due to reduced recursion overhead.
- Recursive Quicksort incurs **function call overhead**, slowing down execution.
- For datasets smaller than **10,000 elements**, both methods perform similarly.

4. Memory Usage in Recursive vs. Iterative Quicksort

Recursive Quicksort uses the **system call stack**, where each recursive call **allocates memory** for function parameters and local variables. If the recursion depth is too high, it may cause a **stack overflow**. In contrast, Iterative Quicksort uses a **fixed-size stack** (`log n`), making it more memory-efficient for large datasets.

5. Impact of Modern Processors on Sorting Speed

Modern processors leverage **branch prediction, caching, and parallel execution**, affecting sorting efficiency.

Processor Optimizations:

- **CPU Cache:** Recursive Quicksort's function calls may lead to frequent memory accesses, reducing cache efficiency.
- **Branch Prediction:** Iterative Quicksort benefits from fewer unpredictable function calls.
- **Multi-threading:** Neither version utilizes parallel execution; parallel sorting algorithms would be more efficient.

Conclusion:

Modern CPUs optimize **iterative Quicksort** more effectively due to reduced recursion overhead and better memory locality.

6. When Recursive Quicksort is Preferable

Although Iterative Quicksort is faster for large inputs, Recursive Quicksort has its advantages:

- **Simpler to implement and read**.
- **Common in standard libraries** like C++ STL, Python's Timsort, and Java's Dual-Pivot Quicksort.
- **If compiler optimizes tail recursion**, it can perform similarly to iterative sorting.

7. Hybrid Sorting Techniques

Many modern sorting implementations use **Hybrid Quicksort**, which combines Quicksort with other algorithms for efficiency:

- **Switching to Insertion Sort for small partitions** (< 16 elements) reduces recursion overhead.
- **Python's Timsort** merges **Merge Sort and Insertion Sort** to optimize sorting operations.

8. Comparison Between Fortran and C for Sorting`

Language	Execution Speed	Memory Management	Optimization
Fortran	Slightly slower than C	Automatic memory management	Optimized for numerical operations
C	Generally faster	Manual memory handling	More aggressive compiler optimizations

Why C is Faster:

1. **Lower function call overhead** (fewer runtime checks).
2. **Explicit memory management** (fewer unnecessary allocations).
3. **More aggressive compiler optimizations** for sorting algorithms.

However, **Fortran is more suited for scientific computing**, making it preferable for numerical operations.

9. Lessons Learned & Challenges Faced

Lessons Learned:

- Understanding Recursive vs. Iterative Algorithms
- Fortran Modules, Subroutines, and File I/O
- Challenges of Stack Usage in Recursive Sorting
- Optimizing Sorting for Large Datasets

Challenges & Fixes:

1. **Handling File I/O in Fortran** → **Fix:** Proper usage of `readUnsorted()` and `writeSorted()`.
2. **Linking Errors & Module Dependencies** → **Fix:** Compile modules separately before linking.
3. **Stack Overflow in Recursive Quicksort** → **Fix:** Limit recursion depth or use an iterative version.

10. Conclusion

This project provided valuable insights into algorithm optimization and Fortran programming. **Iterative Quicksort is more stable for large datasets**, while **Recursive Quicksort is easier to implement but can fail due to deep recursion**. Fortran remains powerful for numerical computing but requires careful memory and performance tuning.

11. Future work

- **Implement a multi-threaded Quicksort for parallel execution.**
- **Compare performance with C++ and Python implementations.**
- **Explore hybrid sorting algorithms** (e.g., switching to Insertion Sort for small subarrays).

Final Comparison:

- **Iterative Quicksort is faster for large inputs (≥ 50,000 elements)** because it avoids recursion overhead and handles memory efficiently.

- **Recursive Quicksort is competitive for small inputs (< 10,000 elements),** where the difference is negligible.
- **If tail recursion optimization is used, recursive Quicksort may close the performance gap.**