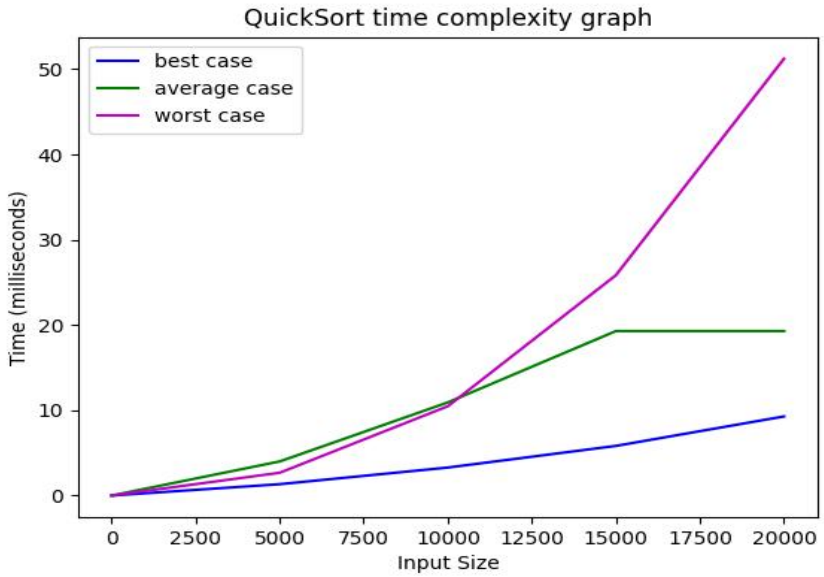
LAB 4 - Sorting algorithm

Report by:
Akash kafle (27)
Nabin kafle (28)

Graph of the QuickSort algorithm's time complexity



Here is the complete breakdown and analysis of the above graph:

Three cases shown: best, average, worst

This graph illustrates the performance of QuickSort under different scenarios.
Understanding these cases is crucial for algorithm analysis and real-world application.

Best case (blue):

- Nearly linear growth, suggesting a time complexity close to O(n log n) which is not immediately seen due to lower data set and vast difference between maximum value and itself.
- This occurs when the pivot chosen always divides the array into roughly equal halves.
- Represents the ideal scenario for QuickSort's performance.

Average case (green):

- Shows logarithmic-like growth, also approximating O(n log n) complexity.
- This is the expected performance in most practical situations.
- Plateaus around 20ms, indicating good scalability for larger inputs.

Worst case (pink):

- Exhibits quadratic-looking growth, suggesting O(n^2) complexity.
- This can occur when the pivot is consistently the smallest or largest element.
- Significantly slower for large inputs, highlighting a potential weakness.

All cases similar for small inputs (< 5000)

- Indicates that for small datasets, the choice of pivot and input order has less impact.
- Suggests QuickSort is efficient for small-scale sorting tasks regardless of data characteristics.

Cases diverge as input size increases

- Demonstrates the growing importance of pivot selection and input order for larger datasets.
- Emphasizes the need for optimizations in large-scale applications.

At 20,000 input size:

- Worst case ~5x slower than best case

LAB 4 - Sorting algorithm

- Best: ~10ms, Average: ~20ms, Worst: ~50ms
- Quantifies the performance difference between cases for a specific large input.
- Helps in estimating real-world performance and setting expectations.

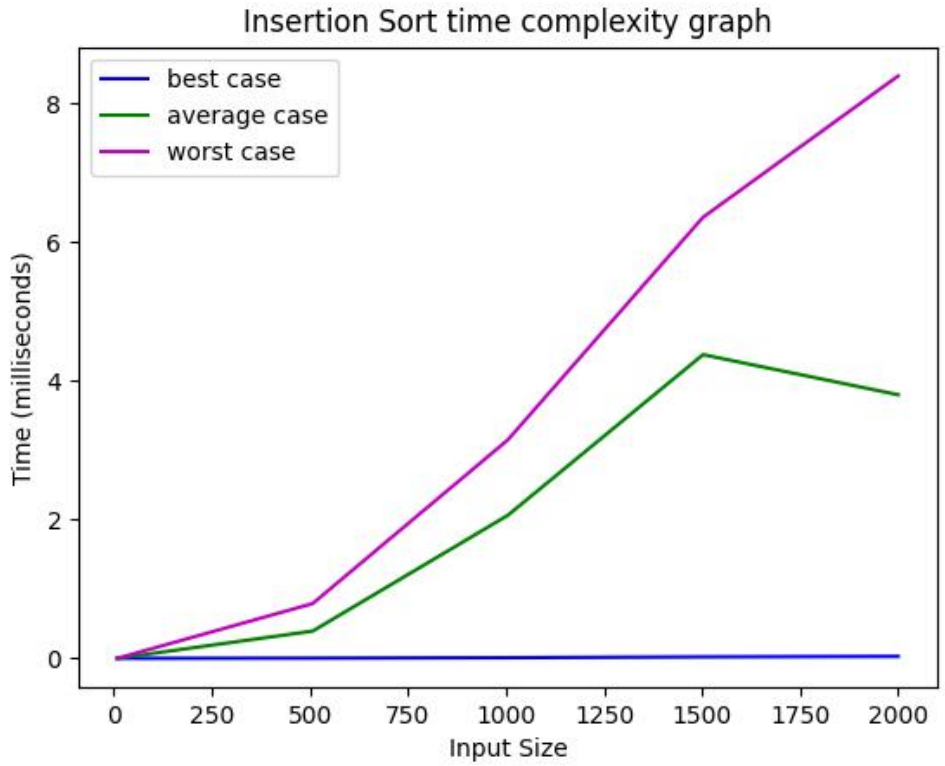
Demonstrates QuickSort's efficiency in typical scenarios

- The average case performance is much closer to the best case than the worst case.
- Explains why QuickSort is often chosen in practice despite its poor worst-case complexity.

Highlights potential issues in worst-case situations

- Warns about possible performance degradation in certain scenarios.
- Encourages consideration of input characteristics when choosing sorting algorithms.

Graph of the Insertion Sort algorithm's time complexity



Here is the complete breakdown and analysis of the above graph:

Time Complexity Cases:

- Best case (blue): Nearly constant, showing minimal growth with input size
- Average case (green): Quadratic growth, increasing significantly with input size
- Worst case (pink): Quadratic growth, slightly steeper than average case

Performance Characteristics:

- Best case: Highly efficient, likely O(n) complexity
- Average and worst cases: Both show O(n^2) complexity
- Significant divergence between best and worst cases as input size increases

Scalability:

- Poor scalability for average and worst cases
- Excellent scalability for best case (nearly sorted data)

Input Size Impact:

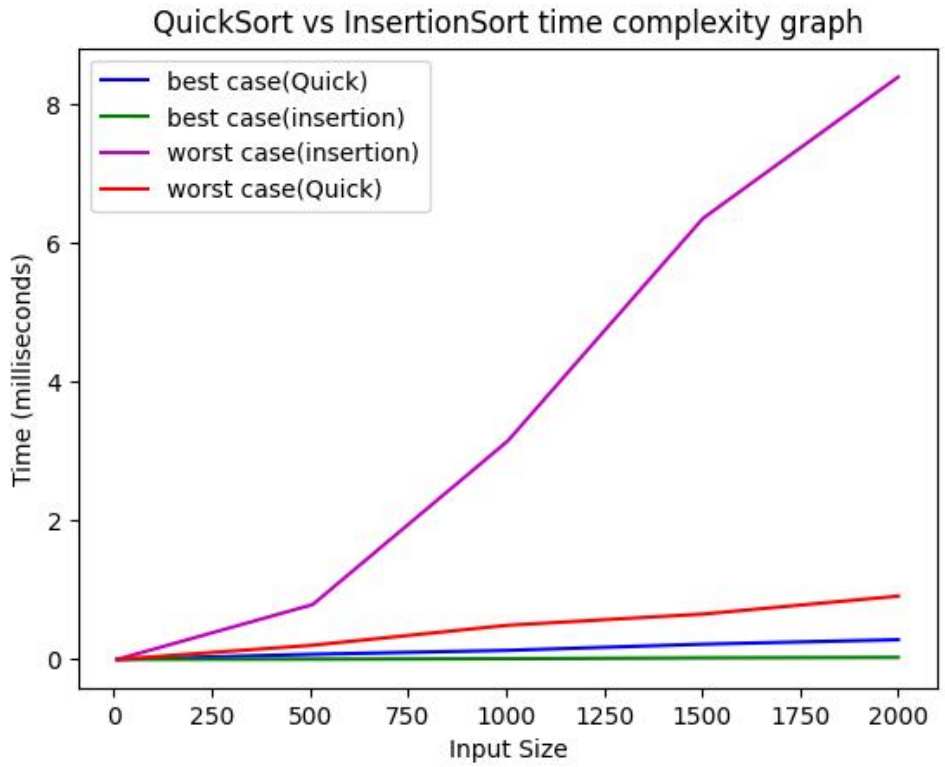
- Small inputs (<500): All cases perform similarly
- Medium inputs (>1000): Drastic performance differences between cases

LAB 4 - Sorting algorithm

Notable Features:

- Average case slightly improves for very large inputs (>1750)
- Best case remains extremely efficient regardless of input size

Graph of the Insertion Sort algorithm's time complexity



Here is the complete breakdown and analysis of the above graph:

Algorithm Comparison:

- QuickSort generally outperforms Insertion Sort
- Insertion Sort's best case is better than QuickSort's best case

Best Case Scenarios:

- Insertion Sort (green): Most efficient overall, likely O(n)
- QuickSort (blue): Very efficient, slightly slower than Insertion Sort's best case

Worst Case Scenarios:

- QuickSort (red): Significantly better than Insertion Sort's worst case
- Insertion Sort (pink): Poorest performance, showing steep quadratic growth

Growth Patterns:

- QuickSort: Both best and worst cases show logarithmic-like growth
- Insertion Sort: Extreme difference between linear best case and quadratic worst case

Crossover Points:

- QuickSort's worst case outperforms Insertion Sort's worst case for all input sizes
- Insertion Sort's best case marginally outperforms QuickSort's best case

Scalability:

- QuickSort: Good scalability in both cases
- Insertion Sort: Excellent scalability in best case, poor in worst case

Practical Implications:

- QuickSort is more reliable for unknown input characteristics

- Insertion Sort could be preferable for small or nearly-sorted datasets

Performance Gaps:

- Largest performance gap is between Insertion Sort's best and worst cases
- QuickSort shows more consistent performance across its cases

Conclusion:

In conclusion, this analysis of QuickSort and Insertion Sort algorithms offers crucial insights into their performance across various scenarios. QuickSort emerges as the more versatile option, showing consistent efficiency across different input sizes and characteristics, making it suitable for most sorting tasks. Insertion Sort, while less efficient for large random datasets, excels with small or nearly-sorted inputs, highlighting its selected utility. The comparison emphasizes that algorithm selection should be based on specific task requirements and expected input characteristics. Understanding these criteria enables developers to make informed decisions, optimizing sorting operations for their particular applications. Ultimately, this analysis underscores the importance of balancing algorithmic efficiency with the specific needs of each use case in computer science and software development.