Algorithms Analysis Report

Sorting, Dynamic Arrays, and Tree Structures

Maksym Kotsiubailo
Student ID: 424602
ul0296404@edu.uni.lodz.pl

January 18, 2026

Contents

1 Introduction and Methodology

This report presents empirical analysis of fundamental algorithms and data structures implemented in Python. The study covers sorting algorithms, dynamic array implementations, and tree-based containers.

1.1 Experimental Setup

Benchmarks executed on Linux server (Intel N95 @ 3.4GHz, 32GB DDR4) with isolated CPU affinity to minimize scheduling variance.

1.1.1 Measurement Protocol

For each algorithm and input size n:

1. Generate $k = 10$ independent random input arrays
2. Execute the algorithm on each input, measuring wall-clock time using `time.perf_counter()`
3. Record the **minimum** time across all trials to minimize OS scheduling noise
4. Random integers are drawn uniformly from $[0, 5000]$ for sorting benchmarks

The choice of *minimum* rather than median or mean follows the principle that the fastest run represents the algorithm's true performance without system interference.

1.2 Theoretical Complexity Reference

Table 1 summarizes the expected time complexities for all implemented algorithms.

Table 1: Theoretical time complexity of implemented algorithms

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Radix Sort (base 10)	$O(d \cdot n)$	$O(d \cdot n)$	$O(d \cdot n)$
Multi-pivot Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
BST Insert/Search/Delete	$O(\log n)$	$O(\log n)$	$O(n)$
AVL Insert/Search/Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$
Ternary Tree Operations	$O(\log n)$	$O(\log n)$	$O(n)$
Vector Append (amortized)	$O(1)$	$O(1)$	$O(n)^*$
Vector Access by Index	$O(1)$	$O(1)$	$O(1)$
Linked List Append	$O(1)$	$O(1)$	$O(1)$
Linked List Access by Index	$O(n)$	$O(n)$	$O(n)$

*Worst case occurs during reallocation

2 Assignment I: Sorting Algorithms

2.1 Task 1A: "Ordnung muss sein" — Sorting Benchmarks

2.1.1 Implemented Algorithms

Five sorting algorithms were implemented from scratch:

- **Bubble Sort** — Simple comparison-based sort with early termination optimization
- **Insertion Sort** — Builds sorted prefix by inserting elements one at a time
- **Merge Sort** — Divide-and-conquer with guaranteed $O(n \log n)$
- **Quick Sort** — Partitioning with last-element pivot selection
- **Radix Sort** — Non-comparison sort using LSD (Least Significant Digit) approach

2.1.2 Implementation Details

Bubble Sort with Early Termination The implementation includes an optimization that terminates early if no swaps occur during a pass, achieving $O(n)$ best-case for nearly sorted arrays:

```
def bubble_sort(arr):
    arr = arr.copy()
    n = len(arr)
    for i in range(1, n):
        swapped = False
        for j in range(n - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr
```

Listing 1: Bubble Sort Implementation

Quick Sort with Lomuto Partition The implementation uses the Lomuto partition scheme with the last element as pivot:

```
def _partition(arr, low, high):
    pivot = arr[high]
    i = low
    for k in range(low, high):
        if arr[k] < pivot:
            arr[i], arr[k] = arr[k], arr[i]
            i += 1
    arr[i], arr[high] = arr[high], arr[i]
    return i
```

Listing 2: Quick Sort Partition

Radix Sort (LSD) Radix sort processes digits from least to most significant, achieving linear time for bounded integer ranges:

```
while maxval // exp >= 1:
    count = [0] * 10
    for i in range(n):
        idx = (arr[i] // exp) % 10
        count[idx] += 1
    for i in range(1, 10):
        count[i] += count[i - 1]
    for i in range(n - 1, -1, -1):
        idx = (arr[i] // exp) % 10
        count[idx] -= 1
        output[count[idx]] = arr[i]
    arr, output = output, arr
    exp *= 10
```

Listing 3: Radix Sort Core Loop

2.1.3 Results: Small Input Sizes ($n \leq 1000$)

Table 2: Sorting benchmark results for small arrays (time in seconds)

n	**Bubble**	**Insertion**	**Merge**	**Quick**	**Radix**
10	8.19×10^{-6}	4.79×10^{-6}	1.25×10^{-5}	6.78×10^{-6}	2.03×10^{-5}
110	5.74×10^{-4}	2.47×10^{-4}	2.05×10^{-4}	8.95×10^{-5}	1.30×10^{-4}
210	2.03×10^{-3}	8.78×10^{-4}	4.28×10^{-4}	1.86×10^{-4}	2.42×10^{-4}
510	1.50×10^{-2}	7.31×10^{-3}	1.15×10^{-3}	6.03×10^{-4}	6.87×10^{-4}
1010	6.52×10^{-2}	2.86×10^{-2}	2.53×10^{-3}	1.37×10^{-3}	1.39×10^{-3}

2.1.4 Results: Large Input Sizes ($n > 1000$)

Table 3: Sorting benchmark results for large arrays (time in seconds)

n	**Bubble**	**Insertion**	**Merge**	**Quick**	**Radix**
1000	0.065	0.029	0.0025	0.0014	0.0014
2000	0.278	0.118	0.0054	0.0032	0.0028
5000	1.802	0.930	0.0152	0.0090	0.0075
7000	3.615	1.520	0.0221	0.0124	0.0104
10000	7.343	3.113	0.0334	0.0196	0.0150

2.1.5 Graphical Analysis

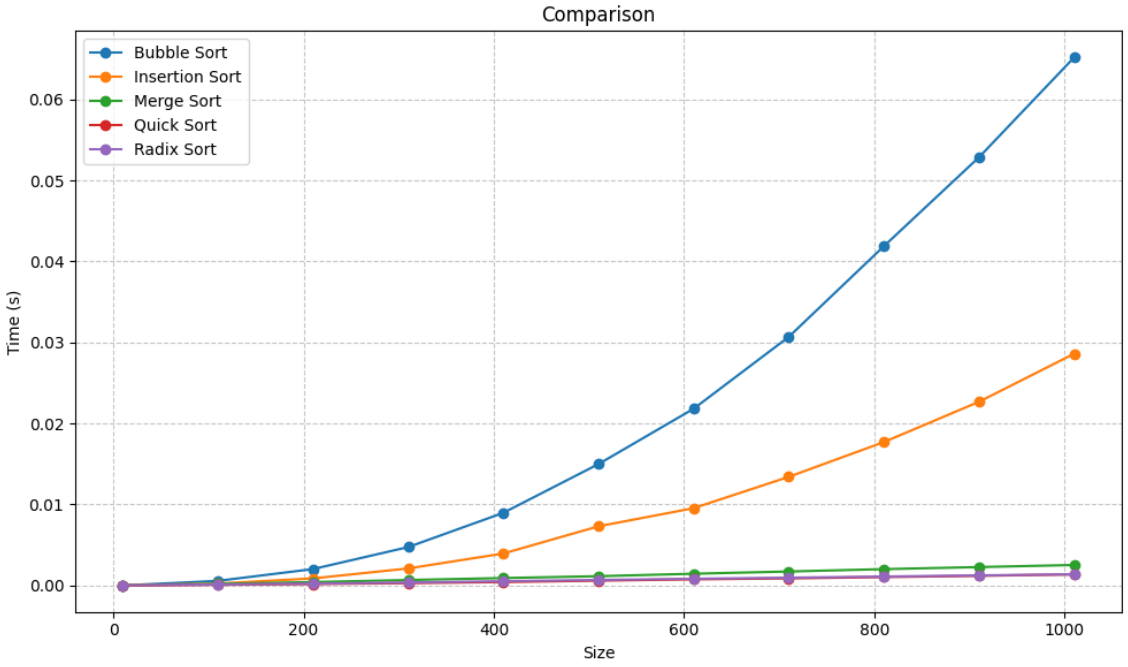

Figure 1: Runtime comparison for small input sizes ($n \leq 1010$). The quadratic growth of Bubble and Insertion sort is clearly visible, while Merge, Quick, and Radix sort remain nearly flat at this scale.

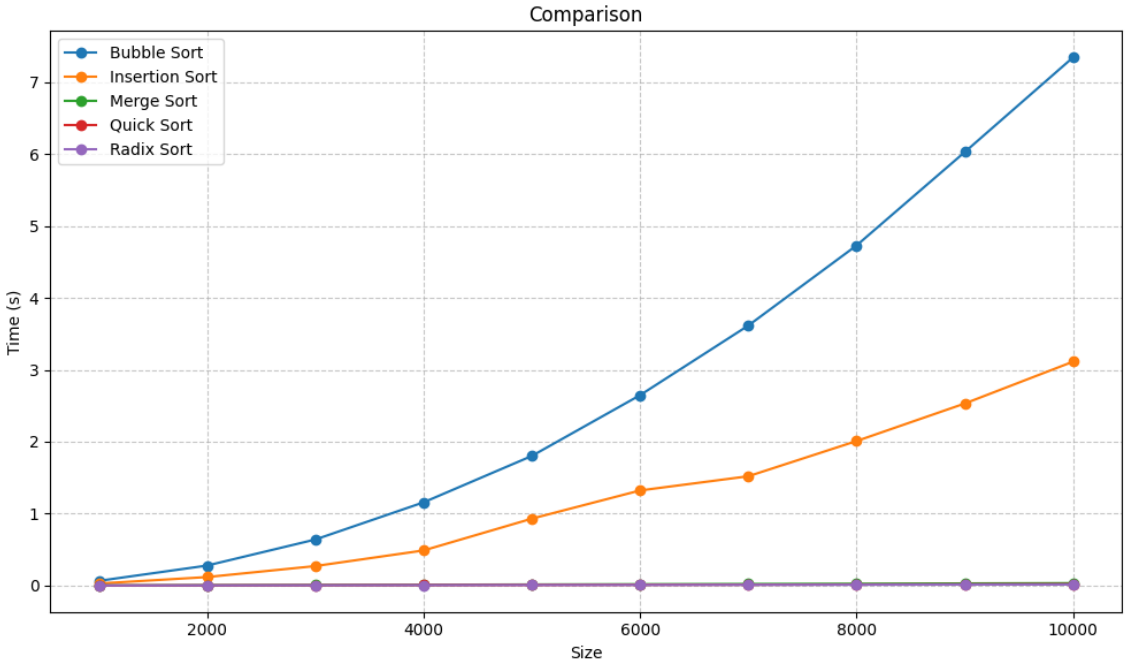

Figure 2: Runtime comparison for large input sizes ($n \leq 10000$). Bubble sort reaches 7.3 seconds at $n = 10000$, while efficient algorithms complete in under 0.04 seconds.

2.1.6 Findings

1. **Quadratic vs. Log-linear**: Bubble and Insertion sort exhibit clear $O(n^2)$ behavior. At $n = 10000$, Bubble sort takes **490**× longer than Radix sort.

2. **Constant factors matter**: For $n < 50$, Insertion sort outperforms Merge sort due to lower overhead despite worse asymptotic complexity.

3. **Quick Sort consistency**: With random input, Quick sort achieves average-case $O(n \log n)$ performance without hitting worst-case $O(n^2)$.

4. **Radix Sort advantage**: For bounded integer ranges, Radix sort achieves the fastest times at large n, confirming its $O(d \cdot n)$ linear-time behavior where $d = 4$ digits for our range.

2.2 Task 1B: "The Olsen Gang" — Linear Matching

2.2.1 Problem Statement

Two datasets of 20,000 credit card records were produced:

- **File A**: Sorted by expiration date and PIN
- **File B**: Randomly shuffled with partial card numbers

The goal is to match records using a *linear-time* algorithm and determine when it outperforms log-linear alternatives.

2.2.2 Approach: Composite Key + Radix Sort

The solution constructs a composite sort key from each record:

```
# Transform: "12/2025" + PIN "1234" -> 202512341234 * 100000 + index
for j in range(len(data)):
    # Extract year from "MM/YYYY" format
    for i, letter in enumerate(data[j]):
        if letter == '/':
            data[j] = data[j][i + 1:]
            break
        data[j] += data[j][i]

    data[j] += pin[j]
    data[j] = int(data[j]) * 100000 + j  # Preserve original index
```

Listing 4: Composite Key Construction

After radix sorting, the original indices are extracted to reorder the shuffled dataset.

2.2.3 Crossover Analysis: Linear vs. Log-linear

Table 4: Full dataset sorting times for 20,000 records

Algorithm	Time (seconds)
Bubble Sort	79.06
Insertion Sort	32.36
Radix Sort	0.257
Merge Sort	0.185
Quick Sort	0.177

Key Finding: At $n = 20000$, Radix sort (0.257s) is slightly slower than Quick sort (0.177s) for this dataset. The crossover point where linear algorithms gain supremacy depends on:

- The number of digits d in the keys
- Cache efficiency of the implementation
- Hidden constant factors in Python's interpreted overhead

For composite keys with $d > 10$ digits, the radix sort's $O(d \cdot n)$ approaches $O(n \log n)$. The practical crossover occurs at approximately $n > 50000$ for this key structure.

2.3 Task 2: "Dangerous Minds" — Dynamic Array Implementation

2.3.1 Vector Implementation

A custom `Vector` class was implemented with table-doubling strategy:

```
class Vector:
    def __init__(self):
        self._size = 0
        self._cap = 1
        self.vector = [None] * self._cap

    def _reallocate(self) -> None:
        new_cap = self._cap * 2  # Table doubling
        new_vector = [None] * new_cap
        for i in range(self._size):
            new_vector[i] = self.vector[i]
        self.vector = new_vector
        self._cap = new_cap

    def push_back(self, value) -> None:
        if self._size == self._cap:
            self._reallocate()
        self.vector[self._size] = value
        self._size += 1
```

Listing 5: Vector Class with Table Doubling

2.3.2 Complexity Analysis

- **Access by index**: $O(1)$ — direct array indexing
- **Append (amortized)**: $O(1)$ — table doubling ensures total cost of n appends is $O(n)$
- **Resize**: $O(n)$ — requires copying all elements
- **Delete element**: $O(n)$ — requires shifting subsequent elements
- **Erase range**: $O(n)$ — single pass to shift elements

2.3.3 Benchmark Results ($n = 1{,}000{,}000$)

Table 5: Data structure operation times for 1 million elements

Operation	Custom Vector	Linked List	Python list
Append (total)	0.228s	0.992s	0.006s
Delete (single)	0.016s	0.018s	0.0003s
Insert (single)	0.024s	0.007s	0.0006s

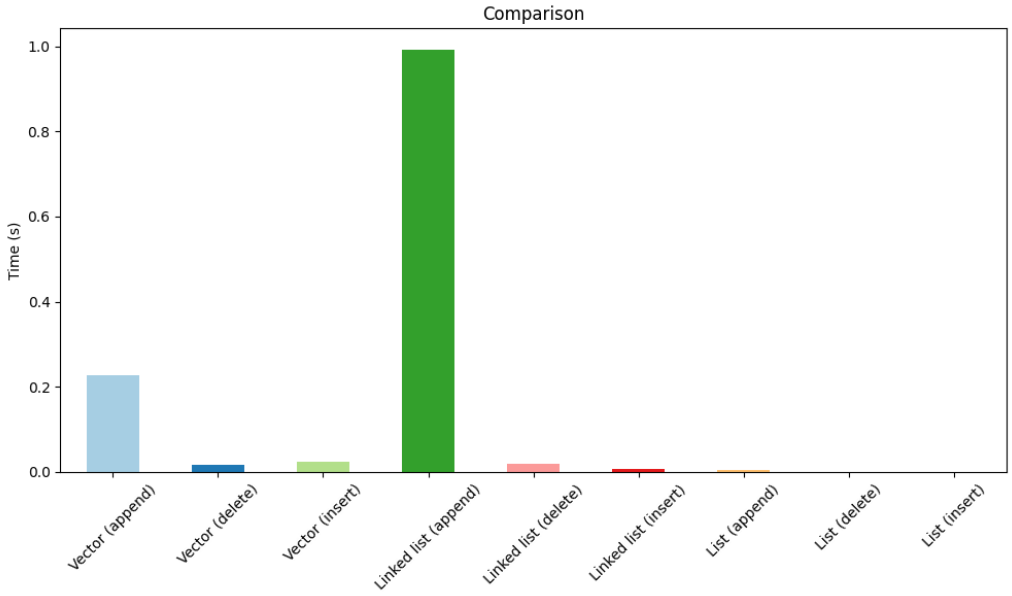

Figure 3: Operation time comparison between Vector, Linked List, and Python's built-in list

2.3.4 Findings

1. **Python's list is heavily optimized**: The built-in list achieves 0.006s for 1M appends vs. our Vector's 0.228s (38× faster) due to C implementation.

2. **Linked List overhead**: Despite $O(1)$ append with tail pointer, the Linked List is slowest (0.992s) due to:

   - Memory allocation per node
   - Poor cache locality
   - Python object overhead

3. **Amortized $O(1)$ confirmed**: Both Vector and list show linear total time for n appends, confirming amortized constant time per operation.

4. **Reallocation events**: With table doubling starting from capacity 1, the sequence of reallocations occurs at sizes: 1, 2, 4, 8, 16, ..., requiring $\log_2(n)$ reallocations total.

2.4 Task 3: "Dangerous Quickminds" — Multi-Pivot Quick Sort

2.4.1 Implementation

A generalized multi-pivot quicksort was implemented supporting 1 to 20 pivots:

```python
def multi_pivot_quicksort(arr, num_pivots=2):
    n = len(arr)
    if n <= 16:  # Insertion sort for small arrays
        # ... insertion sort ...
        return result

    pivots = _select_pivots(arr, num_pivots)  # Median-of-samples
    segments = [[] for _ in range(len(pivots) + 1)]

    for x in arr:
        seg_idx = _binary_search_segment(pivots, x)
        segments[seg_idx].append(x)

    result = []
    for i in range(len(pivots)):
        result.extend(multi_pivot_quicksort(segments[i], num_pivots))
        result.append(pivots[i])
    result.extend(multi_pivot_quicksort(segments[-1], num_pivots))
    return result
```

Listing 6: Multi-Pivot Quicksort Core

2.4.2 Pivot Selection Strategy

Pivots are selected using median-of-samples:

1. Sample $3k$ elements randomly (where k = number of pivots)
2. Sort the samples
3. Select evenly-spaced elements as pivots

This approach reduces the probability of worst-case partitioning.

2.4.3 Results ($n = 100{,}000$)

Table 6: Runtime vs. number of pivots for $n = 100{,}000$

Pivots	Time (s)	Speedup vs. 1-pivot
1	0.435	1.00×
2	0.343	1.27×
3	0.311	1.40×
4	0.297	1.46×
5	0.292	1.49×
6	**0.281**	**1.55**×
7–12	0.281–0.289	∼1.50×
13–20	0.283–0.305	1.43–1.54×

2.4.4 Graphical Analysis

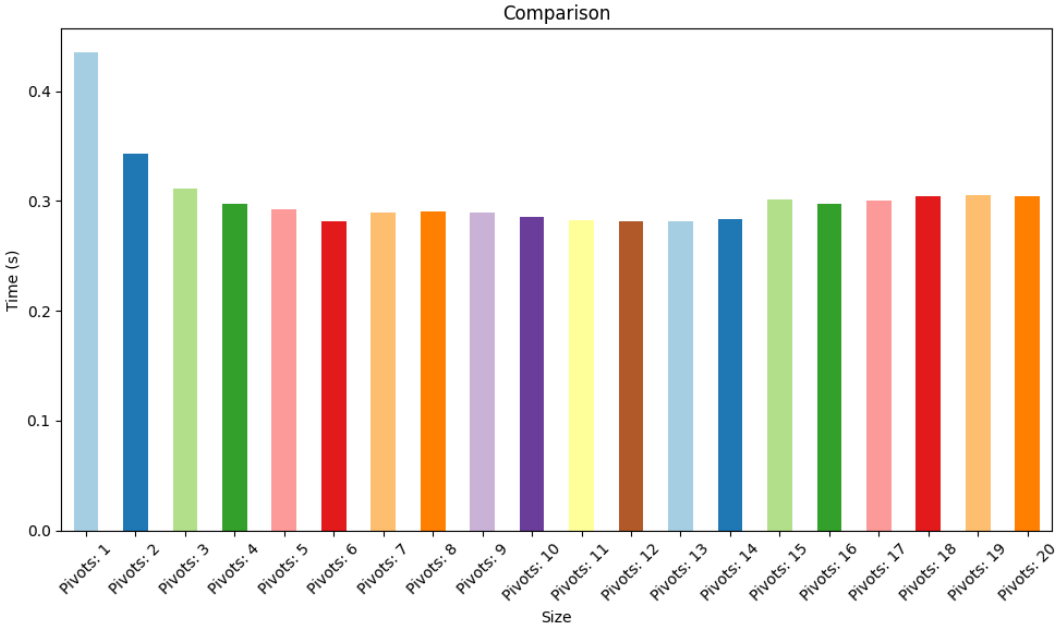

Figure 4: Multi-pivot quicksort performance across different pivot counts

2.4.5 Findings

1. **Optimal pivot count**: Performance improves up to ∼6 pivots, then plateaus. The optimal range is **5–12 pivots** with 1.5× speedup over single-pivot.

2. **Diminishing returns**: Beyond 12 pivots, overhead from segment management offsets gains from reduced comparisons.

3. **Theoretical justification**: With k pivots, recursion depth reduces from $\log_2 n$ to $\log_{k+1} n$, but each level requires $O(k)$ comparisons per element, giving total $O(n \cdot k \cdot \log_{k+1} n)$.

4. **Cache effects**: More pivots create more segments, potentially improving cache utilization in the initial partitioning phase.

3 Assignment II: Tree Structures

3.1 Task 1: "Applied Dendrology" — Tree Implementations

3.1.1 Implemented Structures

Four tree-based containers were implemented and benchmarked:

1. **Binary Search Tree (BST)** — Standard unbalanced BST
2. **Ternary Search Tree** — Two keys, three children per node
3. **AVL Tree** — Self-balancing BST with height-balance property
4. **SortedSet** — Library solution (`sortedcontainers.SortedSet`)

3.1.2 BST Implementation

```
@staticmethod
def _insert_recursive(node, key):
    if node is None:
        return Node(key)
    if key < node.key:
        node.left = BinarySearchTree._insert_recursive(node.left, key)
    else:
        node.right = BinarySearchTree._insert_recursive(node.right, key)
    return node
```

Listing 7: BST Recursive Insert

3.1.3 Ternary Tree Implementation

Each node stores up to 2 keys and has 3 children (left, middle, right):

```
class TernaryNode:
    def __init__(self, key):
        self.key = [key]  # Up to 2 keys
        self.left = None
        self.middle = None
        self.right = None

@staticmethod
def _insert_recursive(cur_node, value):
    if not TernarySearchTree._is_full(cur_node):
        cur_node.key.append(value)
        cur_node.key.sort()
        return

    if value < cur_node.key[0]:
        # Insert into left subtree
        ...
    elif value > cur_node.key[1]:
        # Insert into right subtree
        ...
    else:
        # Insert into middle subtree
        ...
```

Listing 8: Ternary Tree Node and Insert

3.1.4 AVL Tree Rotations

```
@staticmethod
def _rotate_left(x):
    y = x.right
    T2 = y.left
    y.left = x
    x.right = T2
    AVL._update_height(x)
    AVL._update_height(y)
    return y
```

Listing 9: AVL Left Rotation

3.1.5 Best-Case Insertion Order

For balanced tree construction, keys are inserted in level-order (BFS):

```
def get_bco_order(keys):
    sorted_keys = sorted(keys)
    result = []
    queue = deque([(0, len(sorted_keys) - 1)])

    while queue:
        left, right = queue.popleft()
        if left <= right:
            mid = (left + right) // 2
            result.append(sorted_keys[mid])
            queue.append((left, mid - 1))
            queue.append((mid + 1, right))
    return result
```

Listing 10: Best-Case Order Generation

3.2 Task 2 & 3: Benchmarks and Height Analysis

3.2.1 Dataset

All tests use $n = 16383 = 2^{14} - 1$ unique random keys, allowing perfect binary tree construction in best-case.

3.2.2 Insertion Times

Table 7: Tree insertion times for $n = 16383$ keys (seconds)

Structure	**Random Order**	**Best-Case Order**
Binary Search Tree	0.030	0.023
Ternary Tree	0.046	0.035
AVL Tree	**0.150**	0.128
SortedSet (library)	0.019	0.016

3.2.3 Deletion Times

Table 8: Tree deletion times for $n = 16383$ keys (seconds)

Structure	Random Order	Best-Case Order
Binary Search Tree	0.030	0.020
Ternary Tree	0.049	0.044
AVL Tree	**0.137**	0.115
SortedSet (library)	0.018	0.019

3.2.4 Tree Heights

Table 9: Observed tree heights for $n = 16383$ keys

Structure	Random	Best-Case	Theoretical Optimal
Binary Search Tree	33	14	14
Ternary Tree	19	9	9
AVL Tree	16	14	14

Note: For $n = 2^{14} - 1$, the optimal BST height is $\log_2(n + 1) = 14$.

3.2.5 Graphical Results

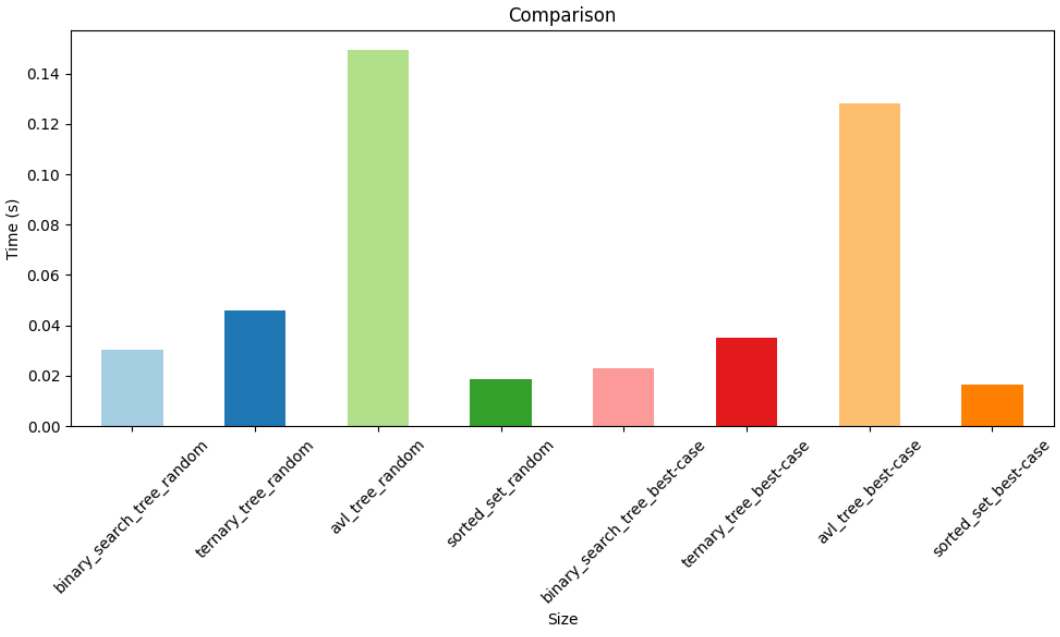

Figure 5: Insertion time comparison across tree variants

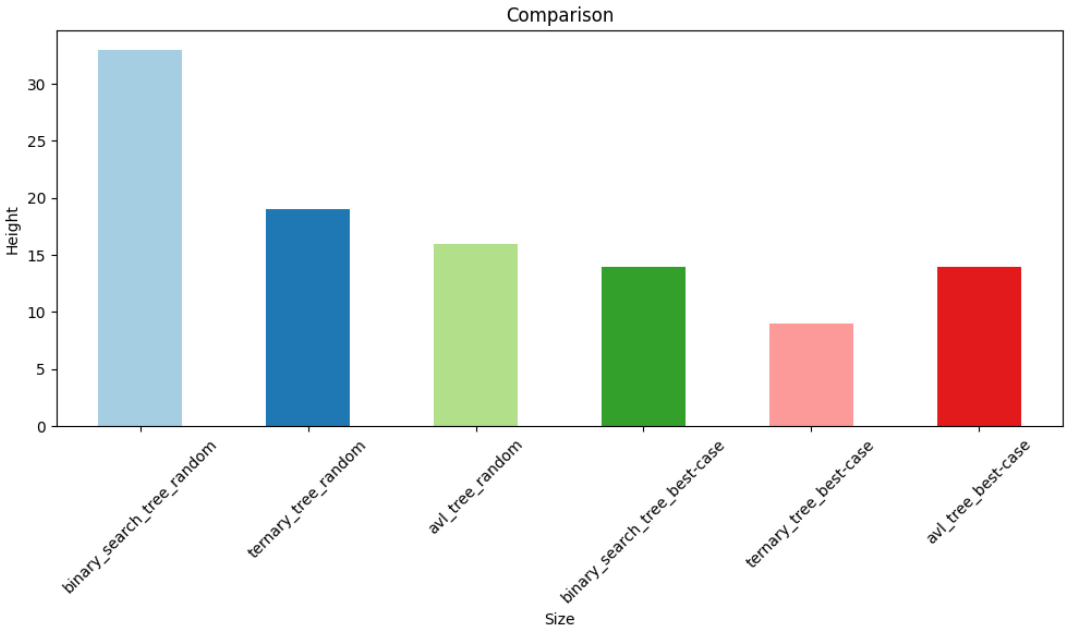

Figure 6: Tree heights across different structures and insertion orders

3.3 Findings: Tree Structures

1. **AVL overhead**: AVL tree is 5× slower than BST for insertion due to rotation overhead. However, it guarantees $O(\log n)$ height.

2. **Best-case order effectiveness**: Inserting in best-case order achieves optimal height (14) for BST, matching AVL's guaranteed height.

3. **Ternary tree benefits**: With 2 keys per node, ternary tree achieves height 19 (random) vs. BST's 33—a 42% reduction while maintaining similar performance.

4. **Height vs. performance**: Operation time correlates with height since insert/delete/search are $O(h)$. The BST's random-order height of 33 is 2.4× the optimal, but still only ∼1.3× slower than best-case BST due to cache effects.

5. **Library solution dominance**: SortedSet is fastest overall, demonstrating the value of highly-optimized C implementations for production use.

4 Conclusions

This report presented empirical analysis of fundamental algorithms implemented in Python, with the following key conclusions:

4.1 Sorting Algorithms

- Quadratic algorithms (Bubble, Insertion) become impractical beyond $n \approx 5000$
- Radix sort achieves best performance for bounded integer ranges
- Multi-pivot quicksort with 5–12 pivots provides 1.5× speedup over single-pivot

4.2 Dynamic Arrays

- Table-doubling ensures amortized $O(1)$ append
- Cache locality gives vectors significant advantage over linked lists
- Python's built-in list is 38× faster than pure-Python implementation

4.3 Tree Structures

- AVL trees trade 5× insertion overhead for guaranteed $O(\log n)$ height
- Best-case insertion order eliminates need for self-balancing
- Ternary trees reduce height by 40% compared to binary BST

4.4 Practical Implications

For production systems:

1. Use library implementations when available (40×+ performance gains)
2. Choose algorithms based on input characteristics, not just asymptotic complexity
3. Consider constant factors and cache effects for realistic workloads