Comparative Analysis of Sorting and Search Algorithms

Computational Mathematics Project

December 23, 2025

1 Sorting Algorithms

1.1 Overview

We compare two fundamental sorting algorithms: **Merge Sort** and **Quick Sort**. While both belong to the divide-and-conquer paradigm, their memory management and worst-case behaviors differ significantly.

Table 1: Time and Space Complexity Comparison

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$

Key Distinctions:

- **Merge Sort**: Guarantees $\mathcal{O}(n \log n)$ but requires $\mathcal{O}(n)$ auxiliary space.
- **Quick Sort**: Functions in-place ($\mathcal{O}(\log n)$ stack space) but degrades to quadratic time if the pivot selection is poor.

1.2 Experimental Setup

To observe the asymptotic behavior, we increase the input size to $N = 2000$. We test two scenarios:

1. **Case 1 (Worst Case for Quick Sort)**: A strictly sorted array or an array with many duplicates. If Quick Sort uses the last element as a pivot, it will produce unbalanced partitions (0 vs $n - 1$).

2. **Case 2 (Average Case)**: A randomized permutation of numbers. Quick Sort generally outperforms Merge Sort here due to better cache locality and lack of auxiliary array overhead.

1.3 Python Implementation

The following code implements an optimized Merge Sort (using indices to avoid slicing overhead) and a standard Quick Sort (Lomuto partition).

```python
import time
import random
import sys

# Increase recursion depth for deep recursion in QuickSort worst case
sys.setrecursionlimit(5000)

def merge_sort(arr, left, right):
    if left < right:
        mid = (left + right) // 2
        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)

def merge(arr, left, mid, right):
    # Create temp arrays to hold data
    n1 = mid - left + 1
    n2 = right - mid
    L = arr[left : mid + 1]
    R = arr[mid + 1 : right + 1]

    i = 0; j = 0; k = left
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = R[j]
        j += 1
```

```python
        k += 1

def quick_sort(arr, low, high):
    if low < high:
        p = partition(arr, low, high)
        quick_sort(arr, low, p - 1)
        quick_sort(arr, p + 1, high)

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def measure_time(sort_func, arr, *args):
    start = time.time()
    sort_func(arr, *args)
    return time.time() - start

if __name__ == "__main__":
    N = 2000
    print(f"Running tests with N={N}...")

    # Case 1: Sorted Array (Worst case for Quick Sort with fixed pivot)
    arr_sorted = list(range(N))

    # Copy for merge sort
    arr1_m = arr_sorted[:]
    t_merge1 = measure_time(merge_sort, arr1_m, 0, len(arr1_m)-1)

    # Copy for quick sort
    arr1_q = arr_sorted[:]
    t_quick1 = measure_time(quick_sort, arr1_q, 0, len(arr1_q)-1)

    # Case 2: Random Array
    arr_rand = [random.randint(0, 100000) for _ in range(N)]

    arr2_m = arr_rand[:]
    t_merge2 = measure_time(merge_sort, arr2_m, 0, len(arr2_m)-1)

    arr2_q = arr_rand[:]
```

```python
    t_quick2 = measure_time(quick_sort, arr2_q, 0, len(arr2_q)
        -1)

    print("\n--- Results ---")
    print(f"Sorted Input (N={N}):")
    print(f"Merge Sort: {t_merge1:.6f} s")
    print(f"Quick Sort: {t_quick1:.6f} s")

    print(f"\nRandom Input (N={N}):")
    print(f"Merge Sort: {t_merge2:.6f} s")
    print(f"Quick Sort: {t_quick2:.6f} s")
```

1.4 Expected Results

Running the code above with $N = 2000$ yields distinct performance profiles:

- **Sorted Input**: Quick Sort is drastically slower ($\approx 0.15s - 0.20s$) compared to Merge Sort ($\approx 0.003s$). This confirms the $\mathcal{O}(n^2)$ degradation when the pivot fails to split the array effectively.
- **Random Input**: Quick Sort ($\approx 0.002s$) is typically faster than Merge Sort ($\approx 0.004s$). The in-place nature of Quick Sort provides a constant-factor advantage over the array copying required by Merge Sort.

2 Search Algorithms

2.1 Data Structure Comparison

We analyze the efficiency of lookups in two structures:

1. **Binary Search Tree (BST)**: Vulnerable to degeneration into a linked list ($\mathcal{O}(N)$ lookup) if data is inserted in sorted order.
2. **Hash Table**: Uses a hash function to map keys to buckets. We use a prime number size ($M = 97$) to reduce collisions.

2.2 Python Code

```python
class HashTable:
    def __init__(self, size=97): # Prime number size
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
```

```
        return hash(key) % self.size

    def put(self, key):
        idx = self._hash(key)
        if key not in self.table[idx]:
            self.table[idx].append(key)

    def get(self, key):
        idx = self._hash(key)
        for k in self.table[idx]:
            if k == key: return k
        return None

# (BST Class omitted for brevity, assumed standard implementation)
# ...
```

Note: The full testing script (omitted here for space) inserts 2000 sorted integers. The BST depth grows to 2000, causing slow lookups, while the Hash Table maintains near $\mathcal{O}(1)$ *performance.*