# Assignment - 2

# Vijaya Krishna Sameeraj Jonnavithula

# Student ID : 005029574

# Course : 2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

# Date : November 3, 2024

1. Asymptotic Analysis and Recurrence Relations :

It is true that Quick Sort and Merge Sort are great options for learning divide-and-conquer algorithms. They are widely utilized in real-world applications and provide fundamental instances of the divide-and-conquer paradigm in action.

They are excellent prospects for the following reasons:

Quick Sort:

Relevance: When effective in-place sorting is needed, Quick Sort is frequently employed. Because of its in-place sorting characteristic and average-case performance, it is particularly well-liked in languages like Python and C++.

Educational Value: By emphasizing instances of both balanced and unbalanced partitioning, Quick Sort sheds light on how pivot choice affects performance. It's also perfect for investigating average and worst-case time complexities.

Merge Sort:  
  
Relevance: Merge Sort is essential for external sorting, such as databases, particularly when dealing with big datasets that are too big to store in memory. For applications that need steady sorting, its predictable performance is essential.

Educational Value: Merge Sort is an excellent illustration of a divide-and-conquer strategy that effectively manages array merging and is appropriate for investigating consistent Θ(𝑛 log⁡𝑛nlogn) performance.

Both algorithms offer a strong basis for researching recurrence relations and their solutions using a variety of techniques, including substitution, recursion trees, and the master theorem, and they demonstrate fundamental ideas in asymptotic complexity analysis.

**Analysis of Quick Sort and Merge Sort :**

**Quick Sort:**

Problem: Sorting an array or list by rearranging its items in either ascending or descending order is done with Quick Sort.

Crucial Actions:

* Select a Pivot: Choose an element from the array to serve as the "pivot"; this can be the first, last, or random element.
* Rearranging the array to place all components larger than the pivot to its right and all elements smaller than the pivot to its left is known as partitioning.
* Sorting Recursively: Sort the subarrays on each side of the pivot using Quick Sort.
* Base Case: The subarray is already sorted, and no more steps are required if it contains zero or one element.

**Merge Sort:**

Issue: Merge Sort also rearranges elements in a list or array in either ascending or descending order.

Crucial Actions:

* Divide: Make two halves out of the array.
* Recursive Sorting: Sort each half using Merge Sort in a recursive manner.
* Merge: To create a single sorted array, combine the two sorted halves by comparing the elements and sorting them in order.
* Base Case: The subarray is deemed sorted and no additional splitting is necessary if it contains zero or one entry.

**Detailed Asymptotic Analysis (O, Ω, Θ Notations) :**

For each algorithm, we analyze the best, worst, and average-case time complexities, giving insight into the performance in various scenarios

**Quick Sort:**

* The best case (Ω) is 𝑂 (𝑛 log ⁡ 𝑛)  
  At each recursion level, O(nlogn) is the result of the pivot splitting the array in approximately equal halves.
* The worst scenario (O) is 𝑂 (𝑛 2)  
  O(n 2) — Occurs when the pivot is always the largest or smallest element, resulting in partitions that are not balanced.
* Average Case (Θ): 𝑂 (𝑛 log ⁡ 𝑛) O(nlogn) — Partitions are often fairly balanced if the pivot selection is random.

**Merge Sort:**

Best, Worst, and Average Case: Θ(𝑛 log ⁡𝑛nlogn) — Because the array is split in half at each level and the merging procedure takes linear time, the time complexity is the same in all situations.

The time complexity of any method is represented by a recurrence relation. The relationships and solutions using various approaches are broken down below.

Quick Sort:

The recurrence relation is as follows: T(n)=T(k)+T(n−k−1)+Θ(n), where k is the partition size.

Methods of Solving the Problem:

Use the substitution method to prove via induction after assuming a solution form and substituting it into the recurrence.

Method of Recursion Trees: Calculate the amount by adding the difficulty of each recursion level and visualizing the costs across them.

Master Method: The recurrence reduces to 𝑇(𝑛) = 2𝑇(𝑛/2) + Θ(𝑛) T(n)=2T(n/2)+Θ(n) if partitioning is balanced, producing Θ(𝑛 log⁡𝑛nlogn).

Merge Sort:

T(n)=2T(n/2)+Θ(n) is the recurrence relation: T(n)=2T(n/2)+Θ(n).

Methods of Solving the Problem:

Substitution Method: To confirm the solution, substitute back, assuming that 𝑇(𝑛) = 𝑛 log ⁡ T(n)=nlogn.

Recursion-Tree Method: Because of halving, each recursion level needs Θ(n) merging effort, with log ⁡𝑛logn levels.

The Master Method This recurrence has the following form: 𝑇(𝑛) = 𝑎𝑇(𝑛/𝑏) + Θ (𝑛) T(n)=aT(n/b)+Θ(n) with 𝑎 = 2 a=2 and 𝑏 = 2 b=2, which results in Θ(𝑛 log⁡𝑛nlogn).

**Practical Implications of Efficiency:**

**Quick Sort:**Use Cases: Because of its average-case performance and efficiency, it is perfect for in-memory sorting. When randomized pivots can be selected to prevent worst-case situations, as is the case with many common library sort methods, Quick Sort is frequently utilized.

Cons: It may not be feasible for data that has already been sorted or almost sorted due to its worst-case performance (Θ(𝑛 2n 2)), unless random pivot selection or other optimizations are used.

**Merge Sort:**

Use Cases: When sorting data on external storage (such a disc), where robust, reliable sorting is crucial, Merge Sort is frequently the algorithm of choice. Because it maintains the order of equal elements, it is also utilized in situations that need for stable sorting.

Cons: In contexts with limited memory, Merge Sort's requirement for extra RAM for merging may be a disadvantage. However, because of its Θ(𝑛 log ⁡𝑛nlogn) performance in every scenario, it continues to be efficient and predictable for big datasets.

1. Implementation and Comparison :

Code run for all cases :   
  
import random

import time

import tracemalloc

# Quick Sort Implementation

def quick\_sort(arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr) // 2] # Middle element as pivot

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

return quick\_sort(left) + middle + quick\_sort(right)

# Merge Sort Implementation

def merge\_sort(arr):

if len(arr) <= 1:

return arr

mid = len(arr) // 2

left = merge\_sort(arr[:mid])

right = merge\_sort(arr[mid:])

return merge(left, right)

def merge(left, right):

result = []

i = j = 0

while i < len(left) and j < len(right):

if left[i] < right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

result.extend(left[i:])

result.extend(right[j:])

return result

# Performance measurement function

def measure\_performance(sort\_function, data):

start\_time = time.time() # Start timing

tracemalloc.start() # Start memory tracking

sorted\_data = sort\_function(data[:]) # Sort a copy of the data to avoid in-place changes

current, peak = tracemalloc.get\_traced\_memory() # Get memory usage

tracemalloc.stop() # Stop memory tracking

end\_time = time.time() # End timing

return {

"time": end\_time - start\_time,

"memory": peak - current

}

# Prepare datasets

n = 10000 # Size of datasets

datasets = {

"sorted": list(range(n)),

"reverse\_sorted": list(range(n, 0, -1)),

"random": [random.randint(0, n) for \_ in range(n)],

}

# Running Quick Sort and Merge Sort on each dataset and recording results

results = {}

for dataset\_type, dataset in datasets.items():

print(f"Running tests on {dataset\_type} dataset")

results[dataset\_type] = {

"quick\_sort": measure\_performance(quick\_sort, dataset),

"merge\_sort": measure\_performance(merge\_sort, dataset)

}

# Print the results for each dataset type

for dataset\_type, metrics in results.items():

print(f"\nResults for {dataset\_type} dataset:")

for sort\_type, performance in metrics.items():

print(f"{sort\_type.capitalize()} - Time: {performance['time']} seconds, Memory: {performance['memory']} bytes")

A screenshot of a computer program

Description automatically generated

Output :

A screenshot of a computer

Description automatically generated

**Analysis of Results :**

Sorted Dataset

* Quick Sort: Time (0.108 seconds), Memory (250,120 bytes)
* Merge Sort: Time (0.187 seconds), Memory (200,128 bytes)

Due to the efficiency of in-place partitioning and the minimal number of swaps needed, Quick Sort outperforms Merge Sort by a small margin on sorted data. However, depending on the pivot selection, this performance may differ.

In this instance, Merge Sort performs predictably but consumes less memory than Quick Sort, possibly as a result of its consistent memory allocation patterns for merging.

Reverse Sorted Dataset

* Quick Sort: Time (0.120 seconds), Memory (250,232 bytes)
* Merge Sort: Time (0.177 seconds), Memory (200,128 bytes)

Although Quick Sort is still quick, it takes a little longer on the reverse-sorted dataset than it does on the sorted one. Reverse sorting may result in the poorest performance in real-world situations, however it appears that this solution has no negative effects on pivot selection.

As anticipated, Merge Sort's performance stays largely consistent. Because it consistently splits and combines every recursion level, its time complexity is constant across dataset kinds.

Random Dataset

* Quick Sort: Time (0.107 seconds), Memory (408,664 bytes)
* Merge Sort: Time (0.240 seconds), Memory (163,820 bytes)

Quick Sort's predicted average-case 𝑂 (𝑛 log ⁡ 𝑛) O(nlogn) time complexity allows it to operate effectively on random data. It uses more memory in this instance, though, maybe as a result of temporary memory allocation during partitioning or recursive stack depth.

Because merging randomly dispersed values is more complicated, Merge Sort takes longer on the random dataset than the others. Merge Sort uses less memory than Quick Sort in this case, suggesting that Quick Sort's recursion may result in higher memory overhead.

**Key Takeaways :**

* Although memory utilisation varies greatly, Quick Sort often runs faster on both sorted and random datasets with our implementation.
* Although Merge Sort has a somewhat longer runtime in random data because of the overhead of merging, it consistently performs reliably in O(nlogn) across dataset types.
* RAM Usage: While Merge Sort is stable, Quick Sort typically uses more RAM when dealing with random data.

**Theoretical vs. Practical Performance**:

**Quick Sort:**

Performance in Theory:

Quick Sort's worst-case time complexity is 𝑂(𝑛 2) O(n 2), while its average-case time complexity is 𝑂(𝑛 log ⁡𝑛) O(nlogn). This worst-case situation usually arises when the pivot is constantly the lowest or largest element, which results in uneven partitions.

Realistic Remarks:

Consistent Speed Across Datasets: Quick Sort's runtime was reasonably constant across sorted, reverse-sorted, and random datasets in our test. This implies that even on sorted and reverse sorted arrays, the pivot selection—in this example, the middle element—avoided the worst-case unbalanced partitions.

Increased Memory Usage on Random Data: Because recursive function calls accumulate memory, Quick Sort's memory usage was higher on the random dataset. Larger datasets may have brief memory spikes as a result of random data's unpredictable recursive depth increases.

* The pivot selection method (picking the middle element) in this case helped to alleviate Quick Sort's theoretical low performance on sorted or reverse sorted data. Such pivot methods, sometimes known as randomised pivots, make Quick Sort fast in most situations by decreasing the chance of encountering the O(n 2) complexity.
* The in-place sorting feature of Quick Sort typically reduces memory usage, but recursion stack depth can result in memory variances, particularly when partition sizes fluctuate, as the random dataset demonstrates.

**Merge Sort:**

Performance in Theory:

In all three scenarios (best, average, and worst), the time complexity of Merge Sort is guaranteed to be O(nlogn) = 𝑂(𝑛 log ⁡𝑛). This consistency results from the fact that, regardless of the input's original order, it always does a complete merging operation and divides the array equally at each stage.

Realistic Remarks:

Consistent Performance Across Datasets: The runtime of Merge Sort remained consistent for both sorted and reverse-sorted data, with a minor increase for random data. Its theoretical complexity, which is unaffected by the order of the data, is consistent with this.

Consistent Memory Usage: Because Merge Sort uses a non-in-place sorting strategy, its memory usage was constant across datasets. Merge Sort unsurprisingly uses more memory since it needs more room for temporary arrays to unite separated subarrays.

* Merge Sort's recursive divide-and-merge methodology, which is independent of data order, accounts for its predictable runtime and memory utilisation. However, compared to Quick Sort, the extra space required for merging may result in worse performance on bigger datasets.
* Although Merge Sort ensures dependable O(nlogn) performance, its extra memory consumption may be a drawback in real-world applications when sorting very big datasets or when memory is limited.

Quick Sort: Generally quick and memory-efficient, however speed can vary depending on the dataset's properties and the pivot selection. Worst-case outcomes are minimized through middle pivot selection or randomization.

Merge Sort is appropriate for datasets that demand dependable performance because it is consistent and stable in all situations. It may lag on big datasets, though, because it needs more RAM.

REFERENCES:   
  
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press.

This is a comprehensive textbook on algorithms that covers the theory and implementation of both Quick Sort and Merge Sort in depth, including analysis of time complexity and practical considerations.

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

This book provides an accessible overview of algorithms with a focus on practical implementation and performance considerations, especially for Quick Sort and Merge Sort.

Skiena, S. (2008). The Algorithm Design Manual (2nd ed.). Springer.

A valuable resource for understanding the real-world applications of algorithms, including an analysis of sorting algorithms’ performance across different data types.

Weiss, M. A. (2012). Data Structures and Algorithm Analysis in C++ (4th ed.). Pearson.

This book provides insights into sorting algorithm efficiency and discusses both theoretical and practical performance across varied input datasets.

Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.