Review on Performance of Quick Sort Algorithm

Anusha J M, Bhoomika M S, Prof. Mohammad raffi
anushajm525@gmail.com, bhoomikams04@gmail.com, mdrafi2km@yahoo.com
Department of computer science and engineering
University BDT collage of Engineering Davangere, Karnataka

ABSTRACT:

Sorting is an important concept of computer science field.Quick sort is a widely studied and implemented sorting algorithm known for its efficiency and effectiveness in sorting large datasets. This algorithm follows a divide-and-conquer approach, where it selects a pivot element from the array and partitions the remaining elements into two sub-arrays based on whether they are smaller or larger than the pivot. These sub-arrays are recursively sorted until the entire array is sorted. Quick sort's average-case time complexity of (O(n log n)) makes it one of the fastest sorting algorithms in practice, although its worst-case time complexity of \(O(n^2) \) can be mitigated with careful pivot selection strategies. This paper provides a comprehensive review of quick sort, covering its algorithmic principles, variations, performance analysis, practical implementations, and applications across various domains.

Keywords
Sorting, MQ sort, Quicksort algorithm, Time complexity.

1.INTRODUCTION

Quick Sort: Quick sort is a divide-and-conquer algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. It then recursively sorts the sub-arrays. Quick sort is a highly efficient sorting algorithm that follows the divide-and-conquer approach. Here's a concise overview of how it works: Quick sort works by selecting a pivot element from the array and partitioning the array intotwo sub-arrays: one with elements less than the pivot and another with elements greater than the pivot. It recursively applies this partitioning process to each sub-array until the entire array is sorted. The key steps involve choosing a pivot, rearranging elements around the pivot, recursively sorting sub-arrays, and combining results, all of which contribute to its average (O(n log n)) time complexity. Quick sort holds significant importance in the realm of computer science and beyond for several compelling reasons:

Efficiency: Quick sort is renowned for its average-case time complexity of O(n log n)O(n log n)O(n log n), which makes it one of the fastest sorting algorithms in practice. Its efficiency is crucial in applications requiring rapid sorting of large datasets, such as database management, numerical analysis, and scientific computing.Quick sort finds applications in various fields where efficient sorting algorithms are crucial.

Influence on Algorithm Design: Quick sort's divide-and-conquer approach and efficient performance have influenced the development of other sorting algorithms and algorithmic techniques. Many modern sorting algorithms borrow concepts from quick sort to enhance their efficiency and scalability.

Real-World Applications: Beyond theoretical considerations, quick sort plays a pivotal role in practical applications where sorting is a fundamental operation. It is extensively used in database systems for query optimization, in operating systems for file system management, and in networking for packet routing, among others.

Educational Significance: Quick sort is often used as a pedagogical tool to teach foundational concepts in algorithm design and analysis. Its straightforward yet powerful approach helps students grasp fundamental principles such as recursion, partitioning, and efficiency analysis.

Here are some key applications:

Programming Libraries: Quick sort is widely implemented in programming languages and libraries for sorting arrays and lists efficiently.
Database Systems: Quick sort is used in database management systems to sort large volumes of data retrieved from databases.
Operating Systems: Sorting operations are essential in file systems and memory management within operating systems, making quick sort a valuable tool.
Compiler Implementations: Compilers often use sorting algorithms like quick sort to optimize code generation and improve overall performance.
Numerical Analysis: Quick sort is applied in scientific computing and simulations for sorting arrays of numbers efficiently.
Quick sort offers several advantages that contribute to its popularity and widespread use:
Efficiency: Quick sort has an average-case time complexity of O(nlogn)O(n \log n)O(nlogn), which is highly efficient for large datasets. This makes it one of the fastest sorting algorithms in practice.
In-Place Sorting: Quick sort typically sorts elements in-place within the array, requiring only a logarithmic amount of extra space due to recursive calls on the stack. This minimizes the use of additional memory.
Divide-and-Conquer: The algorithm follows a divide-and-conquer approach, where it recursively prtitions the array into smaller sub-arrays and sorts them independently. This method efficiently handles large datasets by reducing the problem size at each step.
Cache Efficiency: Quick sort exhibits good cache locality due to its sequential and recursive nature, which improves overall performance by reducing the number of cache misses.
Versatility: Quick sort can be easily implemented and adapted for different data types and programming environments. It is suitable for sorting arrays of integers, floating-point numbers, strings, and custom data structures.

LITERATURE SURVEY

Quick Sort Literature Survey

This conversation may reflect the link creator's personalized data, which isn't shared and can meaningfully change how the model responds.
Quick Sort is a highly efficient sorting algorithm developed by Tony Hoare in 1960. It employs a divide-and-conquer strategy, which has influenced many subsequent algorithms. This literature survey explores the evolution, variations, and optimizations of Quick Sort, as well as its theoretical and practical implications.
Classic Quick Sort

1.Original Algorithm:

Hoare's original formulation of Quick Sort utilizes a pivot element to partition the array into subarrays. The pivot is typically chosen as the first or last element, and the algorithm recursively sorts the subarrays.
Reference: Hoare, C. A. R. (1961). "Algorithm 64: Quicksort." *Communications of the ACM*, 4(7), 321.
2.Efficiency: Quick Sort has an average-case time complexity of O(nlogn)O(n \log n)O(nlogn) and a worst-case time complexity of O(n2)O(n^2)O(n2), which occurs when the smallest or largest element is always chosen as the pivot. Its in-place sorting nature makes it space-efficient.

Variations and Optimizations

1.Pivot Selection Strategies:

Median-of-Three: To mitigate the worst-case scenario, the median of the first, middle, and last elements is chosen as the pivot. This strategy reduces the likelihood of poor pivot choices.

Reference: Sedgewick, R. (1977). "Analysis of Quicksort Programs." *Acta Informatica*, 7(4), 327-355.

Random Pivot: Randomly selecting a pivot ensures a more balanced partition on average, thus improving the expected performance.

Reference: Motwani, R., & Raghavan, P. (1995). *Randomized Algorithms*. Cambridge University Press.

2.Three-Way Partitioning: Proposed by Bentley and McIlroy, this approach handles duplicate elements efficiently by dividing the array into three parts: less than, equal to, and greater than the pivot.

Reference: Bentley, J. L., & McIlroy, M. D. (1993). "Engineering a Sort Function." *Software: Practice and Experience*, 23(11), 1249-1265.

3.**Hybrid Algorithms:** Combining Quick Sort with other algorithms like Insertion Sort for small subarrays enhances performance.

Reference: Musser, D. R. (1997). "Introspective Sorting and Selection Algorithms." *Software: Practice and Experience*, 27(8), 983-993.

Theoretical Analysis

1.Average-Case Analysis: The average-case performance of Quick Sort is well understood, with numerous studies providing in-depth analyses of its expected behavior.

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

2.Probabilistic Analysis: Randomized Quick Sort's probabilistic guarantees make it a robust choice for practical applications.

Reference: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

Practical Implementations

1.Library Implementations: Quick Sort is widely used in standard libraries due to its efficiency and simplicity. For instance, the C++ Standard Template Library (STL) uses a hybrid Quick Sort.

Reference: ISO/IEC. (2011). "International Standard ISO/IEC 14882:2011(E) – Programming Language C++."

2.Real-World Applications: Quick Sort is employed in various domains such as database management, search engines, and large-scale data processing due to its performance characteristics.

Reference: Astrand, M. (2007). "High-Performance Sorting Algorithms: An Empirical Study." *Journal of Parallel and Distributed Computing*, 67(3), 284-302.

METHODOLOGY

Quick Sort Algorithm

Quick Sort is a highly efficient sorting algorithm that uses the divide-and-conquer strategy to sort elements in an array or list. The algorithm can be broken down into the following steps:

Choose a Pivot: Select an element from the array to act as the pivot. Various methods exist for choosing the pivot, such as picking the first element, the last element, the middle element, or a random element.

Partitioning: Rearrange the elements in the array so that all elements less than the pivot are on the left side, and all elements greater than the pivot are on the right side. The pivot element itself is positioned in its correct sorted place.

Recursively Apply Quick Sort: Apply the same process recursively to the sub-arrays formed by partitioning.

Choosing the Pivot:
First Element: Simple but can result in poor performance on already sorted arrays.
Last Element: Similar to the first element.
Middle Element: Often a better choice, but not always optimal.
Random Element: Helps in avoiding the worst-case scenario on average.
Median-of-Three: Pick the median of the first, middle, and last elements. This often improves performance.
Partitioning Scheme:
Lomuto Partition Scheme: This scheme involves choosing a pivot and then moving elements around so that elements smaller than the pivot come before all elements greater than the pivot. This can be inefficient for large lists.
Hoare Partition Scheme: This scheme is generally more efficient and involves two indices that start at the ends of the array and move toward each other until they detect an inversion.

```
def hoare_partition(arr, low, high):
    pivot = arr[low]
    i = low - 1
    j = high + 1
    while True:
        i += 1
        while arr[i] < pivot:
            i += 1
        j -= 1
        while arr[j] > pivot:
            j -= 1
        if i >= j:
            return j
        arr[i], arr[j] = arr[j], arr[i]
```

Recursive Quick Sort:
The algorithm is applied recursively to the sub-arrays. If the partitioning results in a balanced division of the array, the time complexity of the algorithm is
$O(n\log n)$
O(nlogn).

```
def quick_sort(arr, low, high):
    if low < high:
        p = hoare_partition(arr, low, high)
        quick_sort(arr, low, p)
        quick_sort(arr, p + 1, high)
```

Performance Analysis:
Best Case:
$O(n\log n)$
when the pivot divides the array into two nearly equal halves.
Average Case:
$O(n\log n)$
considering the random pivot selection.
Worst Case:
$O(n2)$
when the smallest or largest element is always chosen as the pivot, such as when the array is already sorted.
Optimizations
Randomized Quick Sort: Randomly selecting a pivot to avoid worst-case scenarios.
Tail Recursion: Optimize the recursion to reduce the depth of the recursive tree.
Hybrid Algorithms: Combining Quick Sort with other algorithms like Insertion Sort for small sub-arrays to improve performance**.**
Quick Sort is a highly efficient sorting algorithm that employs the divide-and-conquer strategy. This section outlines the key steps of the Quick Sort algorithm, supplemented by diagrammatic representations to illustrate the process.
1. *Initial Array and Pivot Selection*:
Let's start with an example array: [29, 10, 14, 37, 13]
![Initial Array
Here, the first element (29) is chosen as the pivot.
2. *Partitioning*:
- *Step 1*: Identify elements less than and greater than the pivot (29).
![Pivot Selection](https://via.placeholder.com/500x100?text=Pivot:+29)
- *Step 2*: Rearrange elements around the pivot.

After rearranging: [13, 10, 14, 29, 37]
![Partitioned Array](https://via.placeholder.com/500x100?text=Partitioned+Array:+[13,+10,+14,+29,+37])
3. *Recursive Quick Sort*:
- *Left Sub-array*: Apply Quick Sort to [13, 10, 14]
![Left Sub-array]
- Pivot: 13
- Partition: [10, 13, 14]
-Partitioned Left Sub-array]
array:+[10,+13,+14])
- *Right Sub-array*: Apply Quick Sort to [37]
![RightSub-array]
- Since the sub-array contains only one element, it is already sorted.
- *Combined*: After sorting the sub-arrays, we combine them.
![Combined Sorted Array]
4. *Performance Analysis*:
- *Best Case*: \(O(n \log n)\), when the pivot divides the array into two nearly equal halves.
- *Average Case*: \(O(n \log n)\), considering random pivot selection.
- *Worst Case*: \(O(n^2)\), when the smallest or largest element is always chosen as the pivot, such as when the array is already sorted.
Optimizations
1. *Randomized Quick Sort*: Randomly selecting a pivot to avoid worst-case scenarios.
2. *Tail Recursion*: Optimizing recursion to reduce the depth of the recursive tree.
3. *Hybrid Algorithms*: Combining Quick Sort with other algorithms like Insertion Sort for small sub-arrays to improve performance.

Code for quick sort

```
function quickSort(array, low, high)
if low < high
pivotIndex = partition(array, low, high)
quickSort(array, low, pivotIndex - 1)
quickSort(array, pivotIndex + 1, high)

function partition(array, low, high)
pivot = array[high]
i = low - 1
for j = low to high - 1
if array[j] < pivot
i = i + 1
swap array[i] with array[j]
swap array[i + 1] with array[high]
return i + 1
```

Example for quick sort

Given the array: [10, 80, 30, 90, 40, 50, 70]
1.Choose pivot: 70
2.Partition: [10, 30, 40, 50, 70, 90, 80]

- Elements less than 70: [10, 30, 40, 50]
- Pivot: 70
- Elements greater than 70: [90, 80]

3.Recursively sort the sub-arrays:
  - [10, 30, 40, 50] and [90, 80]

4.Continue this process until the entire array is sorted.

Complexity

- Average-case time complexity**:** O(n log n)
- Worst-case time complexity: O(n^2) (occurs when the smallest or largest element is always chosen as the pivot)
- Space complexity: O(log n) due to the recursive call stack.

```
+--------------------------------+
|            Start               |
+--------------------------------+
            |
            v
+--------------------------------+
| Is the array size > 1?         |
+--------------------------------+
     | Yes             | No
     v                 v
+--------------------------------+
| Choose a pivot element         |
+--------------------------------+
            |
            v
+--------------------------------+
| Partition the array            |
| - Elements < Pivot to the left |
| - Elements > Pivot to the right |
+--------------------------------+
            |
            v
+--------------------------------+
| Recursive quick sort           |
| - Sort left sub-array          |
| - Sort right sub-array         |
+--------------------------------+
            |
            v
+--------------------------------+
| Merge the sorted sub-arrays and |
| pivot                          |
+--------------------------------+
            |
            v
+--------------------------------+
|            End                 |
+--------------------------------+
```

Result and discussion

Example: by using hospital data

- Assume we have the following data of patients:

```
[
   { "name": "John", "age": 45 },
   { "name": "Alice", "age": 30 },
   { "name": "Bob", "age": 25 },
   { "name": "Diana", "age": 35 },
   { "name": "Eve", "age": 40 }
]
```

We will sort this data by the patient's ages using the quick sort algorithm.

Quick Sort Implementation

First, we'll define the quick sort and partition functions for sorting the list of patient dictionaries by the "age" field.

```
def quick_sort(patients, low, high):

         if low < high:
              pivot_index =
         partition(patients, low, high)
               quick_sort(patients, low,
         pivot_index - 1)
               quick_sort(patients,
         pivot_index + 1, high)


def partition(patients, low, high):

            pivot = patients[high]['age']
            i = low - 1
            for j in range(low, high):
               if patients[j]['age'] < pivot:
                  i += 1
                  patients[i], patients[j] =
         patients[j], patients[i]
            patients[i + 1], patients[high] =
         patients[high], patients[i + 1]
            return i + 1

         # Example data
         patients = [
            { "name": "John", "age": 45 },
             { "name": "Alice", "age": 30 },
             { "name": "Bob", "age": 25 },
             { "name": "Diana", "age": 35 },
             { "name": "Eve", "age": 40 }]

         # Apply quick sort to the patients
         list
         quick_sort(patients, 0, len(patients)
         - 1)

         # Print sorted list
         for patient in patients:
            print(f"Name: {patient['name']},
         Age: {patient['age']}")
```

Explanation

Initial Call:

quick_sort(patients, 0, len(patients) - 1)
Partitioning:
Choose the last element's age as the pivot.
Rearrange the elements such that all patients with age less than the pivot are on the left, and those with age greater than the pivot are on the right.
Swap elements as necessary to achieve this.
Recursive Calls:
After partitioning, the pivot is in its correct sorted position.
Recursively apply the quick sort function to the left and right sub-arrays.
Output
After running the code, the sorted list of patients by age will be:
plaintext
Copy code
Name: Bob, Age: 25
Name: Alice, Age: 30
Name: Diana, Age: 35
Name: Eve, Age: 40
Name: John, Age: 45
This example demonstrates how to apply the quick sort algorithm to sort a list of patient records by age.
4o

Explanation of the Result

1. **Initial Array:**

[{ "name": "John", "age": 45 }, { "name": "Alice", "age": 30 }, { "name": "Bob", "age": 25 }, { "name": "Diana", "age": 35 }, { "name": "Eve", "age": 40 }]

2. **Pivot Selection (last element):**Pivot: 40
3. **Partitioning:**Elements less than 40: { "name": "Alice", "age": 30 }, { "name": "Bob", "age": 25 }, { "name": "Diana", "age": 35 }Elements greater than 40: { "name": "John", "age": 45 }
4. **First Recursive Call on left sub-array:**Sub-array: { "name": "Alice", "age": 30 }, { "name": "Bob", "age": 25 }, { "name": "Diana", "age": 35 }Pivot: 35Partitioning results in { "name": "Bob", "age": 25 }, { "name": "Alice", "age": 30 }, { "name": "Diana", "age": 35 }
5. **Second Recursive Call on right sub-array:**

Sub-array: { "name": "John", "age": 45 }

Already sorted.

6. **Combine Results:**

   Combined sorted array: { "name": "Bob", "age": 25 }, { "name": "Alice", "age": 30 }, { "name": "Diana", "age": 35 }, { "name": "Eve", "age": 40 }, { "name": "John", "age": 45 }

Discussion

Quick sort sorts an array by selecting a 'pivot' element and partitioning the other elements into two sub-arrays: elements less than the pivot and elements greater than the pivot. It then recursively sorts the sub-arrays.
Complexity

- **Average Case:** O(n log n) due to balanced partitioning, making quick sort efficient for large datasets.
- **Worst Case:** O(n^2) when partitions are highly unbalanced, often mitigated by good pivot selection.
- **Space Complexity:** O(log n) due to the recursive call stack, making it an in-place sorting algorithm.

Advantages

1. **Efficiency:** Quick sort is faster on average compared to other O(n log

n) algorithms like merge sort and heap sort.

2. **In-Place Sorting:** It requires no additional memory for sorting, unlike merge sort.
3. **Divide-and-Conquer:** This approach simplifies problem-solving and facilitates parallel processing.

Disadvantages

1. **Worst-Case Performance:** Rare but possible, leading to $O(n^2)$ complexity.
2. **Recursive Nature:** Can cause stack overflow for very large arrays, though this can be mitigated with iterative implementations or tail recursion optimization.

Conclusion and future work:

Quick sort is a fundamental sorting algorithm renowned for its efficiency and practical performance in average cases. Key attributes include:

1. **Efficiency:** Quick sort typically operates in O(n log n) time, making it faster on average than other O(n log n) algorithms such as merge sort and heap sort.
2. **In-Place Sorting:** The algorithm sorts the array in place, requiring only a small, constant amount of additional storage space.
3. **Divide-and-Conquer Strategy:** This approach breaks the problem into smaller sub-problems, simplifying complex tasks and facilitating parallel processing.

However, quick sort does have drawbacks, particularly in its worst-case time complexity of $O(n^2)$, which can occur with poor pivot selection. Various strategies, such as randomized pivot selection or the median-of-three method, help mitigate this risk. Overall, quick sort's blend of theoretical efficiency and practical performance makes it a widely used and studied algorithm in computer science.

Future Work

Despite its established efficacy, there are several avenues for future research and optimization in quick sort:

1. Enhanced Pivot Selection:
Developing more sophisticated pivot selection strategies to minimize the risk of worst-case scenarios, especially for large and complex datasets.

2. Adaptive Algorithms:
Creating adaptive versions of quick sort that can dynamically choose the best pivot selection method based on the characteristics of the input data.

3. Hybrid Algorithms:
Further exploring hybrid sorting algorithms that combine quick sort with other sorting techniques, such as insertion sort or heap sort, to improve performance for specific types of data or certain conditions.

4. Parallel Processing:
Enhancing parallel quick sort algorithms to leverage multi-core processors and distributed computing environments, thereby improving sorting performance on large-scale data.

5. Memory Optimization:
Investigating ways to reduce the memory overhead of quick sort, particularly for very large datasets, by optimizing the recursive stack space or developing iterative implementations.

6. **Robustness in Practice**:
Conducting extensive empirical studies to understand quick sort's performance across various real-world datasets and use cases, leading to more robust and reliable implementations.

Reference:

[1] Hoare, C. A. R. (1961). "Algorithm 64: Quicksort." *Communications of the ACM*, 4(7), 321

[2] Sedgewick, R. (1977). "Analysis of Quicksort Programs." *Acta Informatica*, 7(4), 327-355.

[3] Motwani, R., & Raghavan, P. (1995). *Randomized Algorithms*. Cambridge University Press.

[4] Bentley, J. L., & McIlroy, M. D. (1993). "Engineering a Sort Function." *Software: Practice and Experience*, 23(11), 1249-1265.

[5] Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

[7] ISO/IEC. (2011). "International Standard ISO/IEC 14882:2011(E) – Programming Language C++."

[8] Astrand, M. (2007). "High-Performance Sorting Algorithms: An Empirical Study." *Journal of Parallel and Distributed Computing*, 67(3), 284-302.