2024

Assignment 5: Quicksort Algorithm: Implementation, Analysis, and Randomization

- **NAME:** ABDUL RAHEMAN GOTORI
- **STUDENT ID:** 005029919
- **COURSE & TITLE:** ALGORITHMS AND DATA STRUCTURES (MSCS-532-A01)
- **DATE:** 22ND SEPTEMBER 2024

Table of Contents

Deterministic Quicksort

ANALYSIS

Time Complexity Analysis

1. Best-Case & Average-Case Time Complexity

The time complexity of Quicksort is O(nlogn) in both the best-case and average-case scenarios. A logarithmic number of recursive levels is generated as a result of the pivots dividing the array into two nearly equal parts.

2. Worst Case Time Complexity

The time complexity in the worst-case scenario is $O(n^2)$. When the pivot is the smallest or largest element of the array, this results in extremely uneven partitions (one subarray with $n-1$ elements and the other with 0 elements), which degenerates to a performance similar to insertion sort.

Space Complexity

Due to the recursive call stack, the space complexity is O(logn) on average.

When the recursion depth becomes linear, the worst-case space is O(n). For example, when the array is already sorted and the pivot selection results in unbalanced partitions.

- In-place sorting sorts the array in place, necessitating only a fixed amount of additional space for variable storage.
- The algorithm's recursive nature is the primary source of space overhead, as it makes use of the call stack to maintain a list of subarrays that need to be sorted.

Randomized Quicksort

Analysis of Randomization Effects

Randomizing the pivot selection guarantees that the algorithm does not consistently perform poorly on inputs that have already been sorted or reverse sorted. This is a performance improvement. It effectively maintains the average-case time complexity of O(nlogn) by averaging out the pivot choices across all possible elements.

Reduction of Worst-Case Probability - The likelihood of consistently making poor pivot choices (as in the deterministic worst case) is negligible when a pivot is selected randomly. Quicksort is not likely to degrade to its worst-case time complexity, regardless of the input distribution, as a result of this randomization.

Complexity of Expected Time - The expected time complexity remains O(nlogn), but the probability of encountering the $O(n^2)$ worst case has been significantly reduced.

Time Complexity Analysis

Randomized Quicksort introduces randomness into the selection of pivots by selecting a pivot uniformly at random from the subarray that is being sorted. The objective of this modification is to enhance the reliability of performance by preventing the consistent use of poor pivot choices.

Best Case

Time Complexity = O(nlogn)

In a manner similar to deterministic Quicksort, pivots divide the array into two separate, equal halves.

Balanced partitions, similar to the deterministic best case, generate a recursion tree of height logn, with each level requiring O(n) time, resulting in an overall time complexity of O(nlogn).

Average Case

Time Complexity = O(nlogn)

Random pivot selection typically yields partitions that are either balanced or nearly balanced. The randomness guarantees that pivot choices are uniformly distributed across multiple runs. So, the average case time complexity remains O(nlogn).

Worst Case

Time Complexity = $O(n^2)$

The algorithm could still encounter the worst case scenario despite the scenario being highly improbable.

It is theoretically feasible (although exceedingly unlikely) for the pivot to consistently generate partitions that are exceedingly unbalanced across all recursive steps, even with random pivot selection. The likelihood of such an event occurring is exceedingly low, particularly as the input size increases.

Empirical Analysis

```
import random
import time
import sys

# Optional: Increase recursion limit if needed
# sys.setrecursionlimit(100000)

def quicksort(arr):
    """
    Deterministic Quicksort: Always selects the last element as the pivot.
    """
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[-1]  # Choose the last element as pivot
        less = [x for x in arr[:-1] if x <= pivot]
        greater = [x for x in arr[:-1] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

def randomized_quicksort(arr):
    """
    Randomized Quicksort: Selects a random pivot element.
    """
    if len(arr) <= 1:
        return arr
    else:
        pivot = random.choice(arr)  # Choose a random pivot
        less = [x for x in arr if x < pivot]
        equal = [x for x in arr if x == pivot]
        greater = [x for x in arr if x > pivot]
        return randomized_quicksort(less) + equal + randomized_quicksort(greater)

def generate_input(size, distribution):
    """
    Generates input arrays based on the specified distribution.
        """
    if distribution == 'random':
        return random.sample(range(size * 3), size)
    elif distribution == 'sorted':
        return list(range(size))
    elif distribution == 'reverse':
        return list(range(size, 0, -1))
    else:
        raise ValueError("Unsupported distribution type.")

def measure_time(sort_func, A):

    #Measures the execution time of a sorting function.


    Returns:
    - float: Time taken in seconds.
    """
    start_time = time.time()
    sort_func(A)
    end_time = time.time()
    return end_time - start_time

def main():
    input_sizes = [1000, 5000, 10000]  # Reduced input sizes for simplicity
    distributions = ['random', 'sorted', 'reverse']

    for size in input_sizes:
        print(f"\nInput Size: {size}")
        for dist in distributions:
            arr = generate_input(size, dist)

            # Measure time for deterministic Quicksort
            time_det = measure_time(quicksort, arr)

            # Measure time for randomized Quicksort
            time_rand = measure_time(randomized_quicksort, arr)
        print(f"  Distribution: {dist.capitalize()}")
        print(f"    Deterministic Quicksort Time: {time_det:.6f} seconds")
        print(f"    Randomized Quicksort Time:   {time_rand:.6f} seconds")

if __name__ == "__main__":
    main()
```

Observation of results

 Random Distribution

- Both deterministic and randomized Quicksort perform similarly, exhibiting efficient O(nlogn) behavior.

 Sorted and Reverse Distributions

- Deterministic Quicksort shows significantly increased execution times due to unbalanced partitions, approaching $O(n^2)$ time complexity.
- Randomized Quicksort maintains consistent performance close to O(nlogn) by avoiding worst case scenarios through random pivot selection.

Relation to Theoretical Analysis

- The deterministic quicksort algorithm exhibits an average performance of O(nlogn) and a significantly worse performance of O(n2) for specific distributions (sorted and reverse-sorted).

- The advantage of random pivot selection in avoiding worst-case scenarios is reinforced by the consistent O(nlogn) performance of randomized quicksort across all tested distributions.