**Report on Algorithm Efficiency and Scalability: Randomized Quicksort and Hashing with Chaining**

**Introduction**

This report aims to explore the theoretical and empirical performance of two key algorithms: the Quicksort Algorithm with a Randomized Approach and the Hashing Method with an Open Addressing Technique, specifically Chaining. The emphasis is placed on performance and characteristics such as adaptability to input data in these algorithms. Thus, the practical goal in both theory and experiment is to determine how these algorithms work and how effectively they do that, especially in terms of time, which comes with specific input sizes and structures.

**Part 1: Randomized Quicksort Analysis**

**Theoretical Analysis**

The divide-and-conquer algorithm that is applied to sort arrays is known as Quicksort. It has a simple form of operation where an element, called a pivot, is chosen. The array is divided into two parts based on this pivot or element: one consisting of elements less than the pivot and the other consisting of elements more significant than the pivot and repeating the same process on the two parts to sort them. However, its efficiency highly depends on what the pivot is going to be (Kristo et al., 2020). Quicksort with Randomization – In this method, the pivot is picked randomly from all the subarrays to avoid cases such as a worst case, where the sort input is arranged in an order similar to that in the sorted list.

As is the case of deterministic Quicksort, the time complexity of Randomized Quicksort can be stated in terms of the partitioning step's number of comparisons. As with average case scenarios, Randomized Quicksort works in optimal order of time: O (nlogn).

**Why is the Average-Case Time Complexity O(nlogn)?**

Thus, the best way to comprehend the average-case time complexity of the Randomized Quicksort is via the analysis of its recurrence relation that dictates the overall time complexity of the algorithm. In each partitioning step, the array is split into two sub-arrays, and the size of the expected sub-array on average is n/2. Therefore, the recurrence relation can be expressed as:

T(n)=T(n/2)+T(n/2)+O(n)=2T(n/2)+O(n)

This recurring time can be drawn to be O(nlogn), where 'n' depicts the time required to partition at each level of the recursion tree, and the 'logn' factor arises from the height of the recursion tree, which is logn. Randomized Quicksort takes the same principle of operation as Quicksort but chooses the pivot randomly, preventing the algorithm from frequently using the smallest or largest element as the pivot. Consequently, the number of comparisons averages out to be low, and the overall results exhibit O(nlogn) efficiency.

**Empirical Comparison**

To empirically compare the performance of Randomized Quicksort and Deterministic Quicksort, we implemented both algorithms and tested them on a variety of inputs, including:

* Randomly generated arrays
* Already sorted arrays
* Reverse-sorted arrays
* Arrays with repeated elements

To evaluate the practical performance of these algorithms, we recorded the execution times for the same input types for different input sizes and probability distributions.

**Randomly Generated Arrays**

In all attempts made as random arrays, the Randomized Quicksort took less time to sort the elements than the Deterministic Quicksort. If the input given were random, then the execution time of both algorithms would be near the worst case or average case, which is O(nlogn) (Dong et al., 2023). However, the Randomized Quicksort was slightly better because the random choosing of partitions helped it avoid bad partitions that may happen in Deterministic Quicksort, which always uses the first element as the partition-element.

**Already Sorted Arrays**

When applied on already sorted arrays, Deterministic Quicksort was seen to undergo worst-case with its time complexity of O(n2). This is due to using the first element of a sorted array as the pivot in Deterministic Quicksort, which results in partition imbalance. However, in the Randomized Quicksort, we can always achieve O(nlogn) by choosing the pivot points at random, thereby not leading to a degenerate situation.

**Reverse-Sorted Arrays**

The results for reverse-sorted arrays were comparable to those gathered for previously sorted arrays. Deterministic Quicksort was also proven to have been in its worst-case complexity, quadratic time; however, Randomized Quicksort remained in its best-case scenario.

**Arrays with Repeated Elements**

As in the strategy applied in the quicksort algorithm, both implementations worked as expected with many repeat elements in an array. Thus, Randomized Quicksort performs slightly better because a single randomly chosen pivot choice leads to better split partitioning.

**Discussion of Results**

We get results similar to those of predictive analytics insight in the theoretical findings. Randomized Quicksort is again observed to perform significantly better than Deterministic Quicksort when the input list is sorted or nearly sorted. Another scheme used in Randomized Quicksort is the selection of the pivot at random so that, even in the worst determinable case for Deterministic Quicksort, the efficient performance of the randomized form falls within O(nlogn).

The two algorithms worked significantly slower than their average time complexity on random or semi-random instances due to preliminary tests. This brings a case of the pivot selection, whereby it is crucial to guarantee an efficient sorting primarily when the structure of the inputs is specific.

**Part 2: Hashing with Chaining Analysis**

**Theoretical Analysis**

A hash table is an array of records with defined insert, deletion and search methods where the sequence keys map onto the array's slots. If two or more keys hash to the same index then chaining can be employed to store the keys in a chain method, a linked list.

A hash function that distributes keys into the table was adopted in the current implementation. The efficiency of performing operations in the hash table is based on the load factor (λ) which depicts the extent to which the hash table contains elements, i. e., λ = amount of elements/number of slots in the hash table (Wang et al., 2021). In the simplest uniform hashing assumption whereby each key could hash to any slots with the same probability, the expected time on the worst-case insertion, deletion, and search operation is 1 + λ.

**Load Factor and Performance**

When the load factor is small(λ "1), the linked lists in each slot are short and one is capable of being performed in constant time. Hence as load factor increases the lists also tend to increase in size, so as the operations particularly insertion, deletion and search take ‘best and worst ‘case linear time. However, the load factor decides the occupancy of hash table, in order to keep ‘λ’ low, we can reduce or increase the size of the hash table when the load factor reaches a predetermined level.

**Empirical Comparison**

To evaluate the efficiency of our hash table we experimented with different load factors on a range of data sets. The tests included putting, getting, and removing key-value pairs together with the necessity to measure the time taken on each of the processes.

**Insertions**

As predicted, insertions occurred in constant time even at a low load factor such as λ=0. 5 as the majority of keys were placed in slots which were either empty or sparsely occupied. With the load factor higher (e. g. λ=1. 5), insertions were slower by longer chains in every slot indicative of Web sites. Once again, the load factor was kept low after resizing the table; performance was better when the load factor was kept low.

**Searches**

The time for search operations directly correlated with the load factor, where the time was rising linearly with the load factor increase. When load factor was small, searches took the same amount of time, however, when nearing full load factor, longer chains searched slower. The effect of resizing is that the search times also became near constant.

**Deletions**

The performance of deletions was as good as that of searches, because it involves finding the key in the linked list. When the load factor was raised, deletions were slower as the chains, formed by the elements, were longer.

**Discussion of Results**

Using techniques of empirical analysis, we verified the hypothesis that load factor has an enormous impact on the performance of hash table operations as theory suggests. Irrespective of dynamic resizing, which maintains the low balance factor, a hash table requires constant time to make insertions, deletions and lookups. However, if the load factor increases then the size of chains increases at every slot and as a result, the performance of cache decreases.

Another important factor of concern when mapping is the selection of a hash function in order to reduce the chances of collisions while at the same time distributing the keys evenly across the table. On the other hand, the key is to use a good hash function and keep the load factors as low as possible to get the best results of the hash table.

**Conclusion**

In this report, I have evaluated and quantized the performance and growth nature of Randomized Quicksort and Hashing with Chaining. As a result of pivot selection Randomized Quicksort is again seen to outperform Deterministic Quicksort particularly when the input to be sorted is sorted or more structured. Hashing with chaining is good for insertion, deletion, and search when the load factor is kept low by dynamic size. It is therefore clear that both these algorithms highlight that design choices like choosing the pivot and managing the load factor well is critical.

**References**

Dong, X., Wu, Y., Wang, Z., Dhulipala, L., Gu, Y., & Sun, Y. (2023, June). High-Performance and Flexible Parallel Algorithms for Semisort and Related Problems. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (pp. 341-353).

Kristo, A., Vaidya, K., Çetintemel, U., Misra, S., & Kraska, T. (2020, June). The case for a learned sorting algorithm. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data* (pp. 1001-1016).

Wang, J., Chen, S., Dong, L., & Wang, G. (2021). CHTKC: a robust and efficient k-mer counting algorithm based on a lock-free chaining hash table. *Briefings in Bioinformatics*, *22*(3), bbaa063.