# Assignment 3: Understanding Algorithm Efficiency and Scalability

Part 1

Presented Python code evaluates the efficiency of two sorting algorithms: Randomized Quicksort and Iterative Deterministic Quicksort. This study aims to measure the time required by each method for sorting arrays of various sizes. The Randomized Quicksort method is a modified version of the conventional quicksort algorithm. The approach employs a random selection of the pivot element from the array. Randomized\_quicksort is a recursive function that selects a random pivot and partitions the array in a manner that items smaller than the pivot are positioned to the left and bigger elements are put to the right. The random pivot is determined by the randomized\_partition function, which exchanges the chosen pivot with the last element and invokes the partition function. This partition function guarantees the correct rearranging of all items around the pivot, therefore achieving the optimal sorting of the array. Furthermore, Iterative Deterministic Quicksort completely eliminates recursion by using an iterative method. Instead of using recursive calls, this method use a stack to maintain a record of the subarrays that need sorting. The pivot in this scenario is consistently the first element in the subarray, potentially resulting in imbalanced partitions, particularly when the array is already arranged in a sorted or reverse-sorted orientation. The {deterministic\_quicksort\_iterative} function sequentially processes each subarray, using the {deterministic\_partition} function to divide the array based on the initial element and then transfer the subarrays to the stack for further sorting. While this approach circumvents the issue of deep recursion, it may lack efficient sorting if the selection of the pivot is suboptimal.

A white screen with black text

Description automatically generated

A white screen with black text

Description automatically generated

A computer screen shot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

The {compare\_algorithms} function facilitates the comparison of the two algorithms. The methods are evaluated on arrays of sizes 100, 500, and 1000, where each array is randomly generated, sorted, reverse-sorted, and contains repeated entries. The program utilizes the {time.time()} function to quantify the duration of each method. The results are then shown for every possible combination of algorithm and array type. It enables a transparent evaluation of the algorithms' performance in various scenarios. The findings indicate that the performance of both methods is dependent on the size and type of the array. Randomized Quicksort is very efficient, especially across tiny arrays, with an execution time of less than 0.01 seconds for arrays consisting of 100 items. It maintains its efficiency even as the size of the array grows, as seen by the observed timings for arrays with 1000 items. The randomized pivot selection method mitigates the occurrence of worst-case situations, providing more equitable partitions. By comparison, Iterative Deterministic Quicksort exhibits similar performance on smaller arrays but experiences a notable decrease in speed as the size of the array increases, especially for arrays that are already sorted or reverse-sorted. The performance problem stems from the use of the first element as the pivot, which might result in imbalanced partitions. In consequence, the execution time significantly rises for bigger arrays, reaching a maximum of 0.369872 seconds for the reverse-sorted array consisting of 1000 items. In summary, the findings indicate that Randomized Quicksort is both more reliable and efficient, particularly as the size of the array grows. The random selection of pivots allows it to circumvent the inefficiencies seen in the deterministic variant, therefore endowing it with superior suitability for bigger datasets or arrays exhibiting certain patterns, such as pre-sorted or reverse-sorted arrays.

Part 2

The Python code supplied constructs a hash table data structure by using the notion of chaining to manage collisions. A hash table is a commonly used data structure that enables the efficient storing and retrieval of pairs of keys and values. The primary objective of this solution is to provide effective management of key-value pair insertion, search, and deletion processes. The hash table begins with a predefined size of 10 buckets, which may be modified. Each bucket is individually represented by a list. During the initialization of the hash table, a separate empty list is generated for each bucket. The hash table employs a hash\_function to associate keys with certain indices inside the table. This code utilizes Python's inherent {hash()} technique and performs the modulo operation ({%}) to guarantee that the index is within the defined limits of the table size. Thus, the hash function will consistently provide a valid index for storing the key-value combination for every given key. The insertion technique is solely responsible for the addition of key-value pairs. The algorithm first computes the index using the {hash\_function} and then verifies the existence of the key in the bucket corresponding to that index. Upon finding the key, the matching value is promptly updated. If not, a fresh key-value pair is added to the bucket array. Following each item insertion, the hash table computes its load factor, which is the ratio of the number of items to the size of the database. When the load factor above 0.75, the hash table automatically increases its size by double in order to continue operating efficiently. The search technique enables users to locate the value linked to a certain key. The algorithm calculates the index for the key by using the hash function and then finds the matching bucket. Upon finding the key, the corresponding value is provided; otherwise, the method returns {None} to indicate the absence of the key in the table. The delete operation eliminates key-value pairs. This function operates in a manner similar to the search function, first identifying the bucket containing the key and then eliminating the key-value pair from that bucket, provided that it exists. Deleting items results in a proportional drop in the load factor. In order to decide whether the hash table should enlarge itself, the {load\_factor} method computes the current load factor. When resizing, the resize method generates a new hash table that is twice the current size. It then rehashes all the previous entries and distributes them evenly throughout the new, bigger table. This facilitates the preservation of rapid lookups and insertions as the database expands. The 'display' method is used to output the contents of the hash table, displaying each bucket along with the corresponding key-value pairs it contains.   
  
The code in the given example use illustrates the operations of adding, searching, and removing key-value pairs. The insert method appends entries for the ingredients 'apple', 'banana', and 'grape' to the hash table. Upon insertion, the table is visually shown, indicating the storage location of each key-value pair according to the calculated hash. The search for the keys ''apple'} and ''banana'} yields their respective values of 50 and 30, therefore validating the intended functionality of the search process. Ultimately, the {delete} operation eliminates the {'banana'} item, and the revised hash table is produced, indicating that {'banana'} has been effectively eliminated while the other entries remain unchanged. This implementation properly showcases the fundamental actions of a hash table, managing collisions by way of chaining and dynamically adjusting the size to provide optimum efficiency.

A screenshot of a computer

Description automatically generated

A white screen with a white background

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated