Sauhard Shakya

005031397

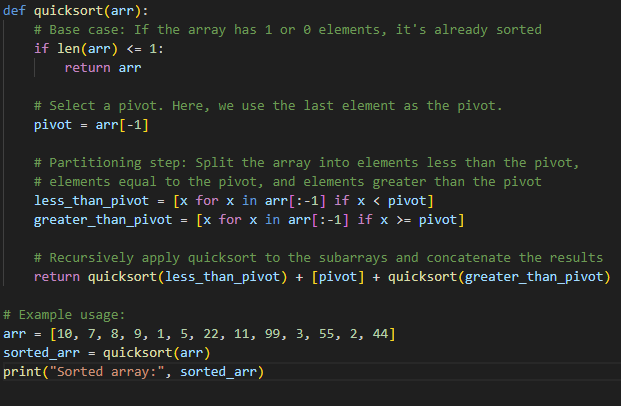
Algorithms and Data Structures (MSCS-532-A01)

Assignment 5: Quick Sort Algorithm

**Quicksort Algorithm**

Like merge sort, quicksort uses divide and conquer and so it’s a recursive algorithm. (Khan) However, quick sort is different than merge sort because in merge sort the divide step does not do anything, everything happens in combine step but in quicksort it is the opposite all real work happens in the divide step.(Khan)

Here is the implementation of a quicksort. In the array, the last element is selected as a pivot. Then it is partitioned into 2 subarrays less than pivot and greater than pivot. The quicksort is applied to both subarrays and combined them with the pivot in the middle.



**Performance Analysis**

**Time Complexity**

1. **Best Case**:
   * The best case occurs when the pivot divides the array evenly into two nearly equal parts at each step. In this case, the recursive depth is proportional to log n, and at each level, we process all nnn elements, leading to an overall time complexity of O(nlogn).
2. **Average Case**:
   * On average, the pivot will divide the array into two reasonably balanced subarrays. In this scenario, the recursion depth is O(log n) and at each level, the array is partitioned and processed in O(n). Therefore, the average-case time complexity remains O(nlogn).
3. **Worst Case**:
   * The worst case occurs when the pivot selection is poor, such as when the pivot is always the smallest or largest element in the array. This results in unbalanced partitions, where one subarray has almost all the elements, and the other is empty. In this case, the recursion depth is O(n)), and at each level, we still process all nnn elements. This leads to a time complexity of O(n^2).

**Explain why the average-case time complexity is \(O(n \log n)\) and the worst-case time complexity is \(O(n^2)\).**

The key reason that the average case is O(n log n) is that, on average, the pivot divides the array into two reasonably balanced subarrays. Over log n recursive calls, we process all n elements at each level.

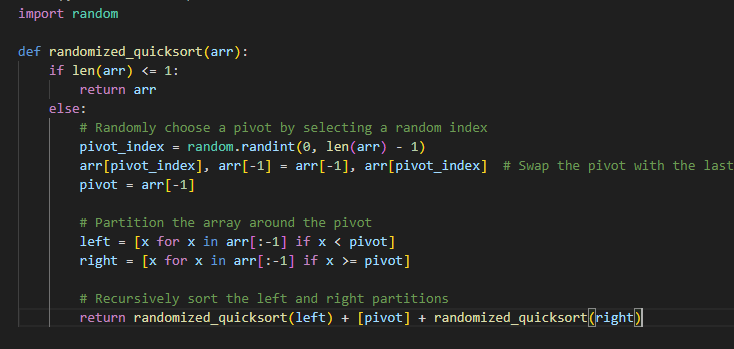
The worst case happens when the pivot selection is consistently poor. For example, if the pivot is always the smallest or largest element, the array will not be partitioned efficiently, leading to unbalanced recursion. In this case, we will have a recursion depth of O(n) and at each depth level, we must process all n elements, resulting in O(n^2) time complexity.

**Discuss the space complexity and any additional overheads associated with the algorithm.**

The space complexity of Quicksort is O(log n) on average because the recursion depth in the average case is O(log n). This is the space required for the function call stack.

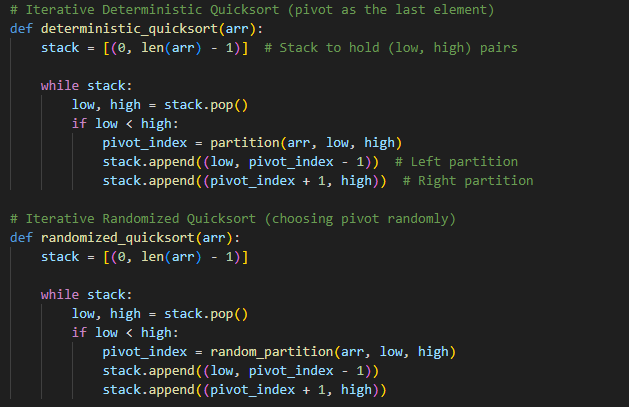
Regarding the additional overheads, this implementation uses extra space for the subarrays created during partitioning. This adds O(n)O(n)O(n) space complexity for each recursive call. Therefore, the total space complexity can be considered O(n) in the worst case, although the recursion stack itself is O(log n) in the best or average case.

**Randomized Quicksort**



Randomization in Quicksort significantly improves its performance by reducing the likelihood of encountering the worst-case scenario, which occurs when the pivot selection consistently divides the array into highly unbalanced subarrays. In the deterministic version of Quicksort, choosing a fixed pivot (such as the first or last element) can lead to inefficient partitions, particularly when the input is already sorted or reverse-sorted, resulting in a time complexity of O(n²). By randomly selecting the pivot, randomized Quicksort ensures that the pivot is unlikely to be consistently the smallest or largest element, leading to more balanced partitions on average and reducing the probability of hitting the worst-case behavior. This results in a typical average-case time complexity of O(n log n), making randomized Quicksort more reliable and efficient, especially for diverse or adversarial input distributions.

**Empirical Analysis**

****

Upon analyzing both deterministic and randomized Quicksort should perform similarly, as the array is already randomly shuffled, so neither pivot selection strategy is prone to degenerate behavior.

The deterministic version (which might pick the first or last element as pivot) will likely perform poorly with a time complexity of O(n²). The randomized version will typically avoid this worst-case scenario, as the random pivot selection reduces the likelihood of repeatedly choosing the smallest or largest element.

The deterministic version again will perform poorly with O(n²) time complexity, as it will consistently select the largest element as the pivot. The randomized version will likely still perform at O(n log n) on average due to the random pivot selection.

Randomized Quicksort offers a more robust solution compared to the deterministic version. While its worst-case time complexity remains O(n²), the likelihood of encountering the worst case is greatly reduced due to the random pivot selection.

Empirical Analysis: The randomized version generally shows improved performance over the deterministic version on sorted or reverse-sorted inputs. On random inputs, both perform similarly, but randomized Quicksort has the advantage of avoiding the worst-case time complexities in edge cases.

**A screenshot of a computer

Description automatically generated**

**Reference:**

GeeksforGeeks. (2023b, September 14). *Quicksort using random pivoting*. https://www.geeksforgeeks.org/quicksort-using-random-pivoting/

Khan Academy. (n.d.). Khan Academy. https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort