Assignment 5: Quicksort Algorithm Implementation, Analysis and Randomization

Pavani Chavali

Department of Computer Science, University of The Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Professor. Brandon Bass

Date: 11/16/2025

Implementation of Quick sort:

```
def pivot(arr, left , right):
    pivot = arr[left] # pivot as first element
    start = left
    end = right

    while start<end:
            while start<=right and arr[start] <= pivot:
                start = start+1
            while end>=left and  arr[end] > pivot:
                 end = end-1

            if start<end:
                arr[start],arr[end] = arr[end],arr[start]

    arr[left], arr[end] = arr[end], arr[left]
    return end

def partition(arr, left , right):
     #partitiong the array
    if left<right:
        loc = pivot(arr,left,right)

        # recursively sort the left subarray
        partition(arr,left, loc-1)

        # recursively sort the right subarray
        partition(arr, loc+1, right)



arr = [2,5,3,3,-1,1]

print(f"original array: {arr}")

partition(arr,0,len(arr)-1)
```

```
print(f"sorted array: {arr}")
```

```
pavanichavali@Pavanis-MacBook-Pro practice % /usr/local/bin/python3 /Users/pavanichavali/Desktop/practice/practice/Quicksort.py
original array: [2, 5, 3, 3, -1, 1]
sorted array: [-1, 1, 2, 3, 3, 5]
pavanichavali@Pavanis-MacBook-Pro practice %
```

2. Performance Analysis:

The time complexity of quick sort mainly depends on selecting pivot on the input array given..

Worst Case: $O(n^2)$

If the array is sorted and the pivot element is taken as first element then the time complexity will be $O(n^2)$. This occurs when the pivot is consistently the smallest or largest element in the subarray. The partition function scans all the given elements (0(n)) and splits the array into [] and [remaining elements]. In the next iteration it does same and 0(n-1) and so on and we will make n recursive calls, and the work done at each call is O(n), O(n-1),O(n-2)and so on. The total work is the sum of the arithmetic series:

$T(n) = O(n)+O(n-1)+O(n-2)+.._O(1) = n+(n-1)+..+1 = n(n+1)/2 = O(n^2)$

Best Case: O(n log n)

This is the ideal case where the pivot is always perfect and this occurs when the pivot is always the median element if the subarray. The partition function does O(n) work and splits the array into two perfectly equal halves, each of size n/2 and next O(n/2) and the other O(n/4). This boils down to the same recurrence relation as Merge sort

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

Average Case: O(n log n):

This is the expected performance for a randomly ordered array or when using a randomized pivot. In this case the split need not be 50/50 to get O(n log n) performance. Any reasonable balanced split will do. The recurrence relation would be T(n) = T(n/10)+T(9n/10)+O(n). Even with this asymmetric split the recursion tree is still O(log n) deep. As long as the split is proportional the depth remain logarithmic and the total complexity remains O(n log n).

The space complexity of Quick sort is driven by the recursion call stack. It's the memory needed to keep track of the partition calls that are waiting to be executed.

Worst Case:O(n) In the worst-case scenario on a sorted array, the recursion tree is completely unbalanced. You have a chain of n recursive calls: partition(0, 6) -> partition(1, 6) -> partition(2, 6) and so on. This requires the call stack to grow to a depth of n, resulting in O(n) space complexity. This can be a serious problem, as it can cause a stack overflow for large arrays.

Best/Average Case: O(log n) In the average case, the recursion tree is balanced. The maximum depth of the recursion will be log n. Therefore, only O(log n) stack frames are needed at any given time.

Overheads:

Unstable: Quick sort is not a stable sort because it does not preserve the relative order of the elements in input. This means if you have two equal elements the original order is not guaranteed after the sorting.

Recursion Overhead: For very small subarrays quick sort is slower so opting Insertion sort is recommended on small arrays. The recursive function calls themselves have a small but non-zero overhead. For very small subarray this overhead makes Quicksort slower than a simple algorithm like Insertion Sort.

Randomized Quicksort:

Implementation:

```
import random

def randomized_quicksort(nums):
    if not nums:
        return nums

    quicksort(nums, 0, len(nums) - 1)
    return nums

def quicksort(nums, low, high):

    if low >= high:
        return

    pivot_index = random.randint(low, high)

    pivot_val = nums[pivot_index]

    left= low
    i = low
    right = high

    while i <= right:
        if nums[i] < pivot_val:
            nums[i], nums[left] =  nums[left], nums[i]
            left += 1
```

```
            i += 1
        elif nums[i] > pivot_val:
            nums[i], nums[right] = nums[right], nums[i]
            right -= 1
        else:
            i += 1

    quicksort(nums, low, left - 1)

    quicksort(nums, right + 1, high)


input1 = [2,5,3,3,-1,1]
print(f"Original array: {input1}")
randomized_quicksort(input1)
print(f"Sorted array:   {input1}")
```

```
pavanichavali@Pavanis-MacBook-Pro practice % /usr/local/bin/python3 /Users/pavanichavali/Desktop/practice/practice/RandomizedQuicksort.py
Original array: [2, 5, 3, 3, -1, 1]
Sorted array:   [-1, 1, 2, 3, 3, 5]
pavanichavali@Pavanis-MacBook-Pro practice %
```

The problem with deterministic Quicksort has a predictable worst case. If any input is already sorted, and your rule is pick the first element you will always get the $O(n^2)$ worst case performance. The algorithms behavior is linked to the input data. By choosing a pivot randomly, Input doesn't matter that is a sorted array is no longer a bad input and the pivot will be chosen from anywhere in the array making a good pivot just as likely as bad pivot. Randomization ensures that on average the pivot will be good enough to split the array into reasonably balanced partitions. This forces the O(n log n) average case behavior. The $O(n^2)$ worst case can still technically happen, but only if the random number generator, by pure chance happens to pic the worst possible pivot at every single

step and the probability of doing so is very less. In short randomization makes Quicksort a reliably fast O(n log n) algorithm in the real world, regardless of the input data.

Empirical Analysis:

The data for different inputs for the deterministic and Randomized Quick sort gives me the understanding that the performance of deterministic Quicksort is dangerously dependent on the inputs structure. It only preforms well if the input is already random. On common, simple inputs like sorted or nearly sorted data performance degrades to $O(n^2)$.This can be overcome by investing small, constant amount of time to pick a random pivot, the randomized quick sort effectively guarantees the O(n log n) average case performance for any input.