Avijit Saha

Student ID #005023281

Algorithms and Data Structures (MSCS-532-B01)

Assignment 5

November 17, 2024

Quick Sort is a sorting algorithm based on Divide and Conquer. It picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. It sorts a list of items in ascending or descending order.

I have selected the first element as a pivot to implement a quick sort algorithm. So, this implementation is a deterministic quick-sort algorithm.

```
def deterministic_partition(self, low, high):
    pivot = self.arr[low]  # Choose the first element as the pivot
    i = low + 1  # Start comparing from the element right after the pivot
```

Figure 1: Pivot Selection

Performance Analysis

From the analysis, the time complexity of Quick Sort for **the average and best case**s is O(n log n), and for the **worst case**, it is O(n^2).

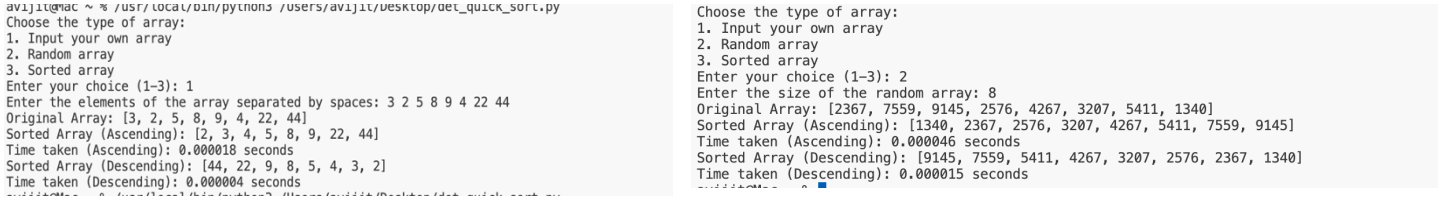

Figure 2: Comparison Between Random and Given Unsorted Array

F**igure 2** shows that the time taken for a given unsorted array is less than the random array for the exact size of an array. In this case, it can be the worst-case scenario because of the random array in deterministic Quick Sort.

Reason for O(n log n) for average case and O(n^2) for Worst Case

Average case: The average case time complexity is O(n log n) because, on average, the algorithm produces reasonably balanced partitions at each step. The array is divided into two

subarrays for the partitioning step, and all array elements are compared with the pivot. Partitioning requires O(n) work. Then, on average, when the pivot splits the array into two subarrays, their size is roughly proportional to n/2. In this case, the recursion depth is approximately log n, as the array size is reduced by half on each recursive call. We can consider the '**Master Theorem'** to solve this recursive relation.

T(n) = aT(n/b) + f(n), where n = size of the input(array), a = number of subproblems(subarray) in the recursion, n/b = size of each subproblem(subarray). All subproblems are assumed to have the same size, f(n) = cost of the work done outside the recursive call, including the cost of dividing the problem and merging the solutions. Here, $a \geq 1$ and $b > 1$ are constants, and f(n) is an asymptotically positive function.

In this case, a = 2, b = 2, and f(n) = O(n). So, the recursion relation for the average case is T(n) = 2T(n/2) + O(n). Since f(n)=O(n^$\log_b$a), the recurrence resolves to T(n) = O (n log n). The average-case time complexity of Quicksort is O(n log n) because the partitioning and recursion form a balanced recursion tree with log n depth, and O(n) work is done at each level.

Worst Case: Quicksort's worst-case time complexity is O(n^2) due to the potential to make inferior choices of pivots that result in very uneven partitions. Here is why:

- **Partitioning Work:** The worst case occurs when the pivot splits the array such that one subarray contains n−1 elements and the other includes 0 or some such highly unbalanced split.
- **Worst-case scenarios:** This happens when the input array is already sorted - ascending or descending- and the pivot is consistently chosen as the first or last element. All elements in the array are the same.

- **Recurrence Relation:** The recurrence for worst case Quicksort is $T(n) = T(n-1)+O(n)$. Here, $T(n-1)$ accounts for sorting the larger subarray with **n-1** elements. $O(n)$ accounts for the partitioning process.
- **Solving the Recurrence:**

  Expand the recurrence:

  $T(n) = T(n-1)+n$

  Substituting, $T(n-1) = T(n-2)+(n-1)$, we get:

  $T(n)=T(n-2)+(n-1)+n$

  So, $T(n) = T(1) + 2 + 3 + \ldots\ldots\ldots\ldots + (n-1) + n$

  The sum of the first integer is $(n(n+1))/2$

  This is how it is $O(n^2)$

In the worst case, Quicksort has a time complexity of $O(n2)$ since unbalanced partitions produce a linear recursion tree of depth n, with $O(n)$ working at each level.

Space Complexity

Quicksort is an in-place sorting algorithm. It uses only a tiny, constant amount of extra storage space. However, since Quicksort is a recursive algorithm, it does use up some space on the call stack. In the **best and average cases**, the height of the recursion tree is $O(\log n)$. In the **worst case**, each recursive call consumes space on the call stack, leading to a total space complexity of $O(\log n)$. In the worst case, the recursion tree degenerates into a linear chain of depth n. Each recursive call adds one more layer to the call stack. Hence, the overall space usage will be $O(n)$.

Randomized Quick Sort: The average-case time complexity of a randomized quick sort is $O(n \log n)$. Each call divides the array or subarray into two parts by selecting a pivot and rearranging

elements. All elements less than or equal to the pivot are on the left, and the more significant elements go on the right side. This is the partitioning, and it takes linear time O(n), where n is the number of elements. Randomized Quicksort avoids the worst case by randomly selecting the pivot, which, in expectation, leads to roughly balanced splits. This means that, on average, each partition splits the array into two parts of approximately equal size. With an average-case split, the recursion tree's recursive levels or depth is proportional to log n. Each level of recursion roughly halves the subarray size. Therefore, with n elements, it takes approximately log n splits to reach subarray size 1, which forms the base case of recursion.

Since the partitioning steps at each level of recursion take O(n) time, and we have O(log n) levels of recursion, the total time complexity is O(n log n) for average-case time complexity. It minimizes the chance of unbalanced partitioning by selecting the pivot randomly, which leads to balanced recursion depth on average.

```
def randomized_partition(self, low, high):
    pivot_index = random.randint(low, high)  # Choose a rand
    self.arr[pivot_index], self.arr[high] = self.arr[high], self.arr[pivot_index]  # Swap pivot to end
    return self.partition(low, high)
```

(parameter) self: Self@QuickSort

Figure 3: Random Pivot Choosing by the random function

```
Choose the type of array:
1. Input your own array
2. Random array
3. Sorted array
Enter your choice (1-3): 2
Enter the size of the random array: 5
Original Array: [5898, 3169, 6814, 167, 9519]
Sorted Array (Ascending): [167, 3169, 5898, 6814, 9519]
Time taken (Ascending): 0.000038 seconds
Sorted Array (Descending): [9519, 6814, 5898, 3169, 167]
Time taken (Descending): 0.000014 seconds
```

```
avijit@Mac ~ % /usr/local/bin/python3 /Users/avijit/Desktop/rand_quick_sort.py
Choose the type of array:
1. Input your own array
2. Random array
3. Sorted array
Enter your choice (1-3): 1
Enter the elements of the array separated by spaces: 6 3 2 4 1 7
Original Array: [6, 3, 2, 4, 1, 7]
Sorted Array (Ascending): [1, 2, 3, 4, 6, 7]
Time taken (Ascending): 0.000084 seconds
Sorted Array (Descending): [7, 6, 4, 3, 2, 1]
Time taken (Descending): 0.000028 seconds
```

Figure 4: Sorting for Random Array vs Given Unsorted Array

From the analysis, randomized quick Sort takes almost similar times for all cases. But it is faster for random arrays. In all cases, reverse sorting takes less time than ascending sorting.

Randomization affects the performance of Quicksort and reduces the likelihood of encountering the **worst-case scenario** when the pivot consistently picks very unbalanced partitions of sizes n−1 and 0. Such a scenario occurs if the input is already sorted or reverse sorted, leading to n recursion levels and quadratic time complexity O(n^2). Randomized selection of the pivot diminishes the chances that poor choices of pivots consistently occur across many recursive calls. Since each pivot is equally likely to split the array at any point, extreme imbalance is far less likely.

Comparison:

```
avijit@Mac ~ % /usr/local/bin/python3 /Users/avijit/Desktop/rand_quick_sort.py
Choose the type of array:
1. Input your own array
2. Random array
3. Sorted array
Enter your choice (1-3): 1
Enter the elements of the array separated by spaces: 6 5 4 3 2 1
Original Array: [6, 5, 4, 3, 2, 1]
Sorted Array (Ascending): [1, 2, 3, 4, 5, 6]
Time taken (Ascending): 0.000080 seconds
Sorted Array (Descending): [6, 5, 4, 3, 2, 1]
Time taken (Descending): 0.000016 seconds
```

Figure 5: Randomized Time Taken.

```
avijit@Mac ~ % /usr/local/bin/python3 /Users/avijit/Desktop/det_quick_sort.py
Choose the type of array:
1. Input your own array
2. Random array
3. Sorted array
Enter your choice (1-3): 1
Enter the elements of the array separated by spaces: 6 5 4 3 2 1
Original Array: [6, 5, 4, 3, 2, 1]
Sorted Array (Ascending): [1, 2, 3, 4, 5, 6]
Time taken (Ascending): 0.000044 seconds
Sorted Array (Descending): [6, 5, 4, 3, 2, 1]
Time taken (Descending): 0.000013 seconds
```

Figure 6: Deterministic Time Taken

I have found some interesting data from the comparison with each other. Deterministic quick Sort takes less time than the randomized quick Sort in all cases, such as randomly generated arrays, already sorted arrays, reverse sorted arrays, and arrays with repeated elements.

```
3. Sorted array
Enter your choice (1-3): 1
Enter the elements of the array separated by spaces: 1 2 2 8 4 3 2 1
Original Array: [1, 2, 2, 8, 4, 3, 2, 1]
Sorted Array (Ascending): [1, 1, 2, 2, 2, 3, 4, 8]
Time taken (Ascending): 0.000060 seconds
Sorted Array (Descending): [8, 4, 3, 2, 2, 2, 1, 1]
Time taken (Descending): 0.000014 seconds
```

Figure 7: Deterministic with Repeated Data

```
avijit@Mac ~ % /usr/local/bin/python3 /Users/avijit/Desktop/rand_quick_sort.py
Choose the type of array:
1. Input your own array
2. Random array
3. Sorted array
Enter your choice (1-3): 1
Enter the elements of the array separated by spaces: 1 2 2 8 4 3 2 1
Original Array: [1, 2, 2, 8, 4, 3, 2, 1]
Sorted Array (Ascending): [1, 1, 2, 2, 2, 3, 4, 8]
Time taken (Ascending): 0.000067 seconds
Sorted Array (Descending): [8, 4, 3, 2, 2, 2, 1, 1]
Time taken (Descending): 0.000019 seconds
```

Figure 8: Random Quick Sort with Repeated Data

But, when I compare it with itself with repeated data, it takes more time than before. It is because of the worst case where O(n^2) applies. For randomized data, the time is still more than that of the deterministic Quick Sort, but compared to the deterministic Quick Sort, the difference is much less with the other types of data. It is because of the average case of time complexity. It chooses the pivot randomly to reduce the time complexity. My analysis shows that deterministic quick sorting takes less time than randomized. But if the size of the array is different or if it has a different structure, such as a repeated array or already sorted array, then it can be time-consuming. So, to be safe, a randomized quick sort is chosen in most cases.

References:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.

Programiz. (n.d.). *Master theorem for divide and conquer recurrence relations*. Programiz. Retrieved November 17, 2024, from https://www.programiz.com/dsa/master-theorem

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.