MSCS 532

Assignment 5

Brendan Yeong
Student ID: 005025797

Quicksort implementation and analysis

Deterministic Quick Sort

Implementation

```python
def partition(arr, low, high):  1 usage  brendanyeong

    # Choose the pivot
    pivot = arr[high]
    i = low - 1

    # Traverse the arr low to high and
    # move all smaller elements to the left side,
    # each iteration the elements from low to i gets smaller
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Move the pivot point after smaller elements and return the position.
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# The quickSort recursive function
def quickSort(arr, low = 0, high = None):  4 usages  brendanyeong
    #set the high index for the first round
    if high is None:
        high = len(arr) - 1

    if low < high:
        # The pivot index is partition and retrieve
        pivot_i = partition(arr, low, high)

        # Recursively calls for small elements for the left side
        # then the bigger elements for the right side.
        quickSort(arr, low, pivot_i-1)
        quickSort(arr, pivot_i+1, high)
```

Performance analysis

Best-case:

The best case occurs when the pivot element selected at each level splits the subarrays into two almost equal-sized subarrays creating one size $[(n-1)/2] \leq n/2$ and one with the size $[(n-1)/2]-1 \leq n/2$. (Corman et al., 2022) This is the best situation where the partitioning is perfectly balanced where at each recursive step, the array is divided into two parts. Then the height of the recursion tree is $\log n$, as the array is halved at each level, where at each level, the partitioning process involves scanning all n elements once. Creating the relation recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

And by using the master theorem we can solve this and thus, giving us a solution of $T(n) = \Theta(n \log n)$.

Average-case:

The average case, Quicksort does not always achieve perfect partitioning, but it generally produces reasonably balanced splits. Lets assume that the pivot divides the array into two subarrays, one with the size of $\frac{1}{3}n$ and other with the size of $\frac{2}{3}n$. This will mean that the number of the levels in the recursion tree is proportional to $\log n$, but the subarrays are not equal in size. Similarly, each level of the recursion still processes all n elements due to the partitioning step. Giving us a $T(n) = n + T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right)$ and solving this will give us $T(n) = \Theta(n \log n)$.

The average case performs closer to the best-case than the worst making Quicksort very efficiently in practice for most inputs.

Worst-case:

The worst-case behavior occurs when the pivot selection is poor. This causes it to partition one subproblem with $n - 1$ elements and one with 0 elements.(Corman et al., 2022) This leads to a recursion tree with n with n levels and at each level all n elements are still be processed. Which gives us a recurrence of:

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

And by using the substitution method we can solve that the recurrence $T(n) = T(n - 1) + \Theta(n)$ has the solution of $T(n) = \Theta(n^2)$.(Corman et al., 2022)

Why the average case time complexity is $O(n \log n)$ and the worst-case time complexity is $O(n^2)$?

This boils down to how the pivot is being selected that made a difference between the average and worst-case. For the average case, the pivot usually divides the array into fairy balanced subarrays, therefore having a logarithmic depth of recursion, giving the complexity of $O(n \log n)$. However, if the pivot selection is consistently poor such as smallest or largest element in the array this could lead to a partition of extremely unbalanced subarrays, thus making it into the worst-case and deteriorating it's performance to a quadratic complexity of $O(n^2)$.

Space Complexity

The space complexity:

Since the partitioning process can be done in-place, meaning that the additional space overhead is minimal. In the best-case, the complexity is $O(\log n)$ since he recursion depth is $O(\log n)$ in the best case. For the worst-case the space complexity is $O(n)$ because the recursion depth can go up to n in the worst case when partitions are highly unbalanced.

Randomized Quick Sort

Implementation

```python
import random

# Randomized partition function for quicksort
def randomized_partition(arr, low, high):  1 usage  brendanyeong
    # Select a random pivot index between low and high
    pivot_index = random.randint(low, high)
    # Swap the pivot with the last element
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    # Call the regular partition function
    return partition(arr, low, high)

# Partition function to rearrange the array
def partition(arr, low, high):  1 usage  brendanyeong
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

```python
# Randomized quicksort function
def randomized_quicksort(arr, low = 0, high = None):  3 usages  brendanyeong
    if high is None:
        high = len(arr) - 1

    if low < high:
        # Randomized partition
        pi = randomized_partition(arr, low, high)
        # Recursively sort elements before and after partition
        randomized_quicksort(arr, low, pi - 1)
        randomized_quicksort(arr, pi + 1, high)

# Wrapper function to simplify usage
def quick_sort(arr):  2 usages  brendanyeong
    randomized_quicksort(arr)
    return arr
```

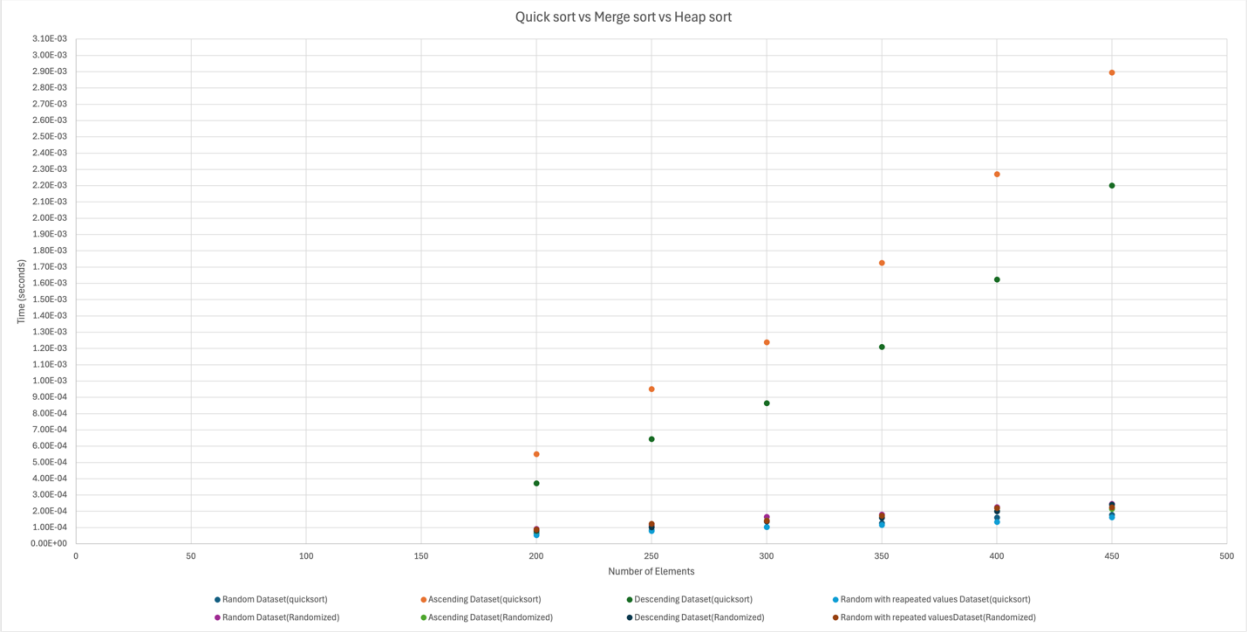
Analysis on performance impact

In Randomized Quicksort, the pivot is chosen randomly from the array. This can be done by swapping a randomly selected element with the last element (or some other designated pivot position) before partitioning. The rest of the algorithm proceeds as in deterministic Quicksort, with the partitioning and recursive sorting of subarrays.

The worst-case time complexity of deterministic Quicksort, $O(n^2)$ occurs when the pivot is poorly selected at each step. By randomizing the pivot selection, the chance of always choosing the worst possible pivot such as, the smallest or largest element is greatly diminished. In randomized Quicksort, the probability of encountering a worst-case partition at every step is very low. Hence, the average performance is much closer to the best-case scenario, $O(n \log n)$, even in cases where deterministic Quicksort might have performed poorly (Cormen et al., 2009).

On average, randomized Quicksort performs as well as deterministic Quicksort, with an expected time complexity of $O(n \log n)$.The randomness ensures that the algorithm behaves efficiently regardless of the input, as poor pivot selections are rare and unlikely to occur repeatedly throughout the sorting process (Cormen et al., 2009).

Empirical analysis



Discussion of Results:

- Random and Random with repeated values arrays: Both deterministic and randomized Quicksort will exhibit $O(n\ log\ n)$ performance, but Randomized Quicksort might have a slight overhead due to the randomness.

- Sorted and Reverse-Sorted Arrays: Deterministic Quicksort will likely degrade to $O(n^2)$ performance on these arrays, while Randomized Quicksort will maintain $O(n\ log\ n)$ complexity by avoiding consistently poor pivot choices.

Randomized Quicksort ensures better performance on average across all types of input arrays, especially when input arrays are sorted or reverse-sorted, where deterministic Quicksort is prone to worst-case behavior.

Reference:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509