5511 hw4

Tony Siu

October 2024

1 Sorting Algorithms

1.1 Problem

The problem is that given an array A of unordered integers, and the size N of the unordered array of integers, we are to sort the array of integers in ascending order from the left to right.

- **Inputs:** The input to the algorithm is an array A of n unordered integers, and an integer N representing the size of the array
- **Outputs:** As the algorithm should sort the array in place, there is not output from the algorithm.

Below are the bubble sort, insertion sort and selection sort algorithms done recursively.

Algorithm 1 Recursive Bubblesort code here

```
procedure BubbleSort($A$, $n$)
    if $n$ is None then
        $n \leftarrow$ length of $A$        ▷ Set $n$ to the size of the array if not provided
    end if
    if $n = 1$ then
        return
    end if
    BubbleSortRecursive($A$, $n - 1$)
end procedure

procedure BubbleSortRecursive($A$, $n$)
    if $n = 1$ then
        return
    end if
    for $i = 0$ to $n - 2$ do
        if $A[i] > A[i + 1]$ then
            Swap $A[i]$ and $A[i + 1]$ ▷ Swap if the current item is greater than the next
        end if
    end for
    BubbleSortRecursive($A$, $n - 1$)     ▷ Recursively sort the first $n - 1$ elements
end procedure
```

Algorithm 2 Recursive Insertion Sort code here

```
H
    procedure InsertionSort($A$, $n$)
        if $n \leq 1$ then
            return
        end if
        InsertionSortRecursive($A$, $n$)
    end procedure

    procedure InsertionSortRecursive($A$, $n$)
        if $n \leq 1$ then
            return
        end if
        InsertionSortRecursive($A$, $n - 1$)
        $last \leftarrow A[n-1]$
        $j \leftarrow n - 2$
        while $j \geq 0$ & $A[j] > last$ do
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j - 1$
        end while
        $A[j+1] \leftarrow last$
    end procedure
```

Algorithm 4 Recursive Selection Sort code here

```
    procedure SelectionSort($A$, $n$)
        $end \leftarrow$ length of $A$
        if $n \geq end - 1$ then
            return
        end if
        SelectionSortRecursive($A$, $n - end$, $end$)
    end procedure

    procedure SelectionSortRecursive($A$, $start$, $end$)
        if $start \geq end - 1$ then
            return
        end if
        $minIndex \leftarrow start$
        for $i = start + 1$ to $end - 1$ do
            if $A[i] < A[minIndex]$ then
                $minIndex \leftarrow i$
            end if
        end for
        Swap $A[start]$ with $A[minIndex]$
        SelectionSortRecursive($A$, $start + 1$, $end$)
    end procedure
```

2 Lomuto Partition VS Hoare Partition [2][1]

Quicksort is a divide-and-conquer algorithm that sorts an array by partitioning it into subarrays around a pivot element. The efficiency of Quicksort largely depends on the partition scheme used.

2.1 Lomuto Parittion

Algorithm 5 Lomuto Partition

```
function PARTITION(A, p, r)
    $x = A[r]$                                   ▷ pivot
    $i = p - 1$
    for $j = p$ to $r - 1$ do
        if $A[j] \leq x$ then
            $i = i + 1$
            EXCHANGE($A[i], A[j]$)
        end if
    end for
    EXCHANGE($A[i + 1], A[r]$)
    return $i + 1$
end function
```

The total comparisons of the Lomuto partition scheme is $C_{Lomuto}(n) = n - 1$ where each A[j] is compared with the pivot. The worst-case for the exchange operation is $S_{Lomuto}(n) = n$, up to n - 1 swaps during the loop and 1 final swap of the pivot. The best-case swaps for $S_{Lomuto}(n) = 1$ when all elements are greater than the pivot. The total number of operations $T_{Lomuto}(n)$ can be expressed as $T_{Lomuto}(n) = T(k)+T(n-k-1)+(n-1)$ comparisons $+S_{Lomuto}(n)$ exchanges. K is the number of elements less than or equal to the pivot. The Lomuto partition function contributes $O(n)$ operations during quicksort.

2.1.1 Lomuto Worst Case

In the worse case, an already sorted array, the pivot divides the array in to sizes n - 1 and 0:

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

Unrolling the recurrence gives us:

$$\begin{aligned} T_{Lomuto}(n) =& T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &\vdots \\ &= T(1) + 2 + 3 \ldots + n \qquad = \frac{n(n+1)}{2} \end{aligned} \tag{1}$$

In summary, the Lomuto partitioning for quicksort gives us a $O(n^2)$ number of operations for the worst case.

2.1.2 Lomuto Average Case

Assuming the pivot divides the array roughly in half gives us $T_{Lomuto}(n) = 2T(\frac{n}{2})+n$. Simplifying the recurrance relation gives us $T_{Lomuto}(n) = nlog_2n+n$ which is approximately $T(_{Lomuto}(n)) \approx nlogn$. The average case time complexity is $\theta(nlogn)$

2.2 Hoare Partition

Algorithm 6 Hoare Partition

```
function HOARE-PARTITION(A, p, r)
    $x = A[p]$                                        ▷ pivot
    $i = p - 1$
    $j = r + 1$
    while True do
        repeat
            $j = j - 1$
        until $A[j] \leq x$
        repeat
            $i = i + 1$
        until $A[i] \geq x$
        if $i < j$ then
            EXCHANGE($A[i], A[j]$)
        else
            return $j$
        end if
    end while
end function
```

Each increment/decrement may involve multiple comparisons due to the inner loops. The average comparisons is $C_{Hoare}(n) \approx \frac{n}{2}$. Swaps occur only when A[i] and A[j] are out of place. Average swaps is $S_{Hoare}(n) \approx \frac{n}{2}$. This forms a recurrance relation of the below where $low \leq k \leq high$ and Dn represents the number of operations in the partition function with smaller constants than in Lomuto's method:

$$T_{Hoare}(n) = T(k) + T(n-k) + Dn$$

2.2.1 Worst Case Analysis

Even in the worse, Hoare's partition does not degrade as severely compared to Lomuto because it always makes at least one element swap to the oppossite side.

$$T_{Hoare}(n) = T(n-1) + Dn$$

This forms the recurrance for worst case below which gives us total operations of $T_{Hoare}(n) = D \cdot \frac{n(n-1)}{2}$ and a time complexity of $O(n^2)$ operations but with a smaller constant D compared to Lomuto:

$$T_{Hoare}(n) = T(1) + D(n-1) + D(n-2) + \ldots + D1$$

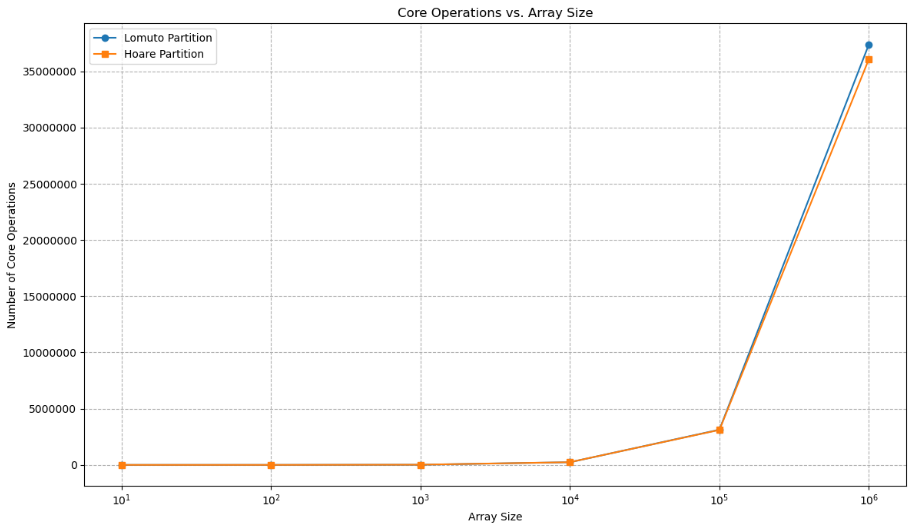
2.2.2 Average Case Analysis

Assuming a fair split, the average case gives us;

$$T_{Hoare}(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + Dn$$

Simplifying for a recurrance relation gives us the below, total operations $T_{Hoare}(n) \approx Dnlogn$ and time complexity of $O(nlogn)$ operations with a smaller constant compared to the Lomuto Partition scheme;

$$T_{Hoare}(n) = Dnlog_2 n + n$$



As depicted above via empirical experiments, Hoare's partition scheme is more efficient than Lomuto's due to fewer comparisons and swaps per partition. Below lists the stdout of the results depicted in the above graph. While both algorithms have $O(nlogn)$ average case complexity, the smaller constants in Hoare's method result in fewer total operations T(n). In practice, quicksort with Hoares partition out performs Lomuto partition especially for large datasets. This is my code implementation for the empirical plot.

Lomuto:

- Sizes: [10, 100, 1000, 10000, 100000, 1000000]
- Core Operations:[38.4, 987.4, 16959.8, 238076.6, 3157125.2, 37336879.2]

Hoare:

- Sizes: [10, 100, 1000, 10000, 100000, 1000000]
- Core Operations: [63.8, 1199.6, 18713.4, 236285.6, 2975252.2, 36152114.4]

References

[1] R.L. & Stein C. Cormen T.H. Leiserson C.E. Rivest. Introduction to Algorithm (3rd ed.) MIT Press. 2009.

[2] R. Sedgewick. *Implementing quicksort programs.* Communications of the ACM. 21(10),847-857. 1978.