Homework Assignment: Recursive Algorithms and Partition Efficiency

Mauricio Estrella

October 1, 2025

1 Recursive Sorting

To sort an array of integers, write recursive version of (a) Bubble Sort, (b) Selection Sort, and (c) Insertion Sort, respectively.

(a) Recursive Bubble Sort works by completing one pass of the standard bubble sort (which moves the largest element to its correct final position at the end of the array) and then recursively calling the function on the rest of the array (i.e., from the first element to the second-to-last element).

- **Input**: An array of integers (A) and an integer n representing the portion of the array to be considered for sorting (from index 0 to n-1).
- **Output**: The function sorts the array in-place. The original array is modified.
- **Relationship**: The output is a permutation of the input array (A) where for any two indices i and j, if $i < j$, then $A[i] <= A[j]$.

Algorithm 1 Recursive Bubble Sort

```
   Function BubbleSort($A$, $n$)
1: if $n <= 1$ then
2:     return
3: for $i \leftarrow 0$ to $n - 2$ do
4:     if $A[i] > A[i + 1]$ then
5:         exchange $A[i]$ with $A[i + 1]$
6: BubbleSort($A$, $n$)
```

(b) Recursive Selection Sort works by finding the minimum element in the unsorted portion of the array and placing it at the beginning of that portion. The function is then called recursively for the rest of the array.

- **Input**: An array of integers (A), the total size of the array (n), and the startIndex for the current unsorted subarray.
- **Output**: The function sorts the array in-place. The original array is modified.
- **Relationship**: The output is a permutation of the input array (A) where for any two indices i and j, if $i < j$, then $A[i] <= A[j]$.

Algorithm 2 Recursive Selection Sort

```
   Function SelectionSort($A$, $n$, $startIndex$)
1: if $startIndex >= n - 1$ then
2:     return
3: $minIndex = startIndex$
4: for $i \leftarrow startIndex + 1$ to $n - 1$ do
5:     if $A[i] < A[minIndex]$ then
6:         $minIndex = i$
7: exchange $A[startIndex]$ with $A[minIndex]$
8: SelectionSort($A$, $n$, $startIndex + 1$)
```

(c) Recursive Insertion Sort works by first recursively sorting the first n-1 elements, and then inserting the n-th element into its correct position within the already sorted part of the array.

- **Input**: An array of integers (A) and an integer n representing the portion of the array to be considered for sorting (from index 0 to n-1).
- **Output**: The function sorts the array in-place. The original array is modified.
- **Relationship**: The output is a permutation of the input array (A) where for any two indices i and j, if $i < j$, then $A[i] <= A[j]$.

Algorithm 3 Recursive Insertion Sort

```
Function InsertionSort($A$, $n$)
1: if $n <= 1$ then
2:     return
3: InsertionSort($A$, $n$)
4: $lastElement = A[n-1]$
5: $j = n-2$
6: while $j >= 0$ and $A[j] > lastElement$ do
7:     $A[j+1] = A[j]$
8:     $j = j-1$
9: $A[j+1] = lastElement$
```

2 Partition Time Efficiency Analysis

Compare the time efficiency of Lomuto Partition and Hoare Partition using methods we have covered and try to be as accurate as possible.

To accurately compare the time efficiencies of Lomuto and Hoare Partitions, we must do a probabilistic analysis. We start with assumptions:

- **Assumption**: Every possible permutation of the input array is equally likely.

Based on this assumption, we can calculate the expected number of times that a key operation (like a swap) will occur. We focus on swaps when comparing these two algorithms because we can determine that we perform roughly the same comparisons in both algorithms. For Lomuto, we perform n - 1 comparisons as seen on scratch paper (line 4). For Hoare, we perform approximately n + 1 comparisons (lines 5 and 6). The pointers i and j start at opposite ends and stop when they cross. In total, every element (except the pivot) will be visited by either i or j exactly once. The total number of comparisons is approximately n + 1. Swaps are analyzed in the following sub sections.

2.1 Lomuto Analysis

For each element A[j], we compare it to the pivot x. If the input is truly random, there is roughly a 50% probability that A[j] will be less than or equal to the pivot. Therefore, the swap operation inside the loop is expected to run on about half of the n-1 elements, leading to an average of (n-1)/2 swaps.

- **Best Case**: 0 swaps. This occurs if all elements are greater than the pivot.
- **Worst Case**: n-1 swaps. This occurs if the array is sorted in reverse and the pivot is the smallest element.
- **Average Case**: (n-1)/2 swaps

2.2 Hoare Analysis

The condition for a swap is stricter. A swap only occurs when the left pointer finds an element A[i] that belongs on the right side and the right pointer finds an element A[j] that belongs on the left side. A more formal probabilistic analysis (as found in the textbook Introduction to Algorithms) shows that this happens far less frequently. The result is an average of approximately n/6 swaps.

- **Best Case**: 0 swaps. This occurs on an already-sorted array.

- **Worst Case**: floor(n/2) swaps. This occurs when the array is structured to maximize misplaced pairs. A perfect example is a reverse-sorted array.
- **Average Case**: n/6 swaps.

2.3 Testing Results and Analysis

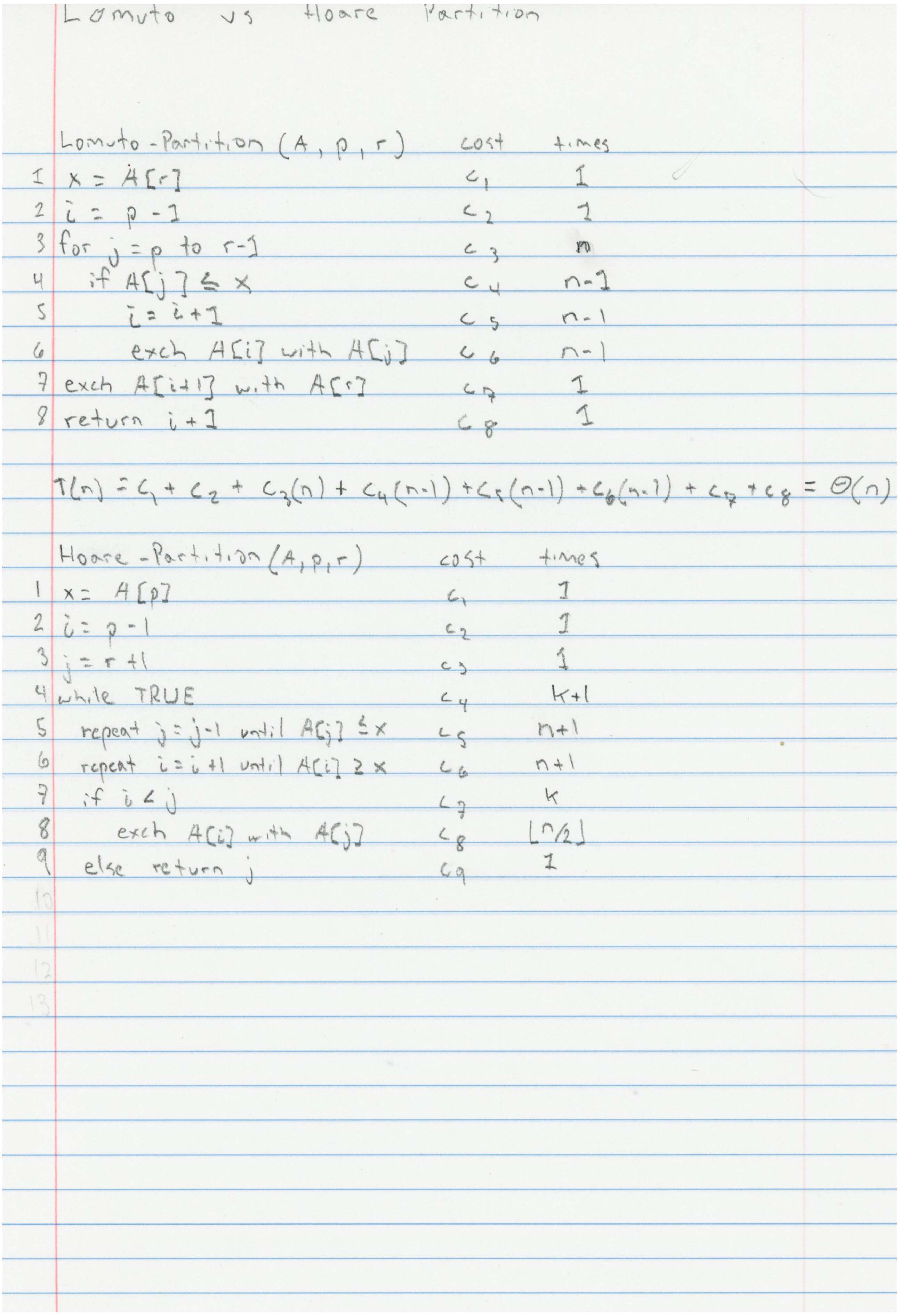
Above, we conceptualize why the Hoare partition should do better than the Lomuto partition on average. Now, we run actual tests on a variety of different inputs. We structure testing to show empirical evidence of the above theoretical analysis. We add swap counters to our algorithms and time it's execution. We choose random data, sorted data, reverse-sorted data and data with many duplicates to test both partitions.

```
Array Size: 10000
Number of Trials: 50

==============================================================================================
--- Results (Averages) ---
==============================================================================================
Data Type              |   Lomuto Time (ms) |    Hoare Time (ms) |    Lomuto Swaps |     Hoare Swaps
----------------------------------------------------------------------------------------------
Random Data            |             1.4442 |             0.8728 |           5,305 |           1,771
Sorted Data            |             1.8633 |             0.6465 |          10,000 |               0
Reverse-Sorted Data    |             0.6487 |             0.6213 |               1 |               1
Many Duplicates        |             1.2828 |             0.8688 |           6,291 |           1,969
----------------------------------------------------------------------------------------------
```

As you can see above, on completely random data, Hoare performs faster and does roughly 3x less swaps than Lomuto. This is where Hoare stands out. An interesting result that differed from what we expected was seeing a swap count of 1 for a reverse-sorted array when we stated above that this could cause a worst case of floor(n/2) swaps. As we can see, because we choose the pivot as the first element in our implementation, we only perform 1 swap.

3 Attachments



Lomuto vs Hoare Partition

```
  Lomuto-Partition(A, p, r)                cost     times
1 x = A[r]                                 c_1      1
2 i = p - 1                                c_2      1
3 for j = p to r-1                         c_3      n
4     if A[j] ≤ x                          c_4      n-1
5         i = i + 1                        c_5      n-1
6         exch A[i] with A[j]              c_6      n-1
7 exch A[i+1] with A[r]                    c_7      1
8 return i + 1                             c_8      1
```

$$T(n) = c_1 + c_2 + c_3(n) + c_4(n-1) + c_5(n-1) + c_6(n-1) + c_7 + c_8 = \Theta(n)$$

```
  Hoare-Partition(A, p, r)                 cost     times
1 x = A[p]                                 c_1      1
2 i = p - 1                                c_2      1
3 j = r + 1                                c_3      1
4 while TRUE                               c_4      k+1
5   repeat j = j-1 until A[j] ≤ x          c_5      n+1
6   repeat i = i+1 until A[i] ≥ x          c_6      n+1
7   if i < j                               c_7      k
8       exch A[i] with A[j]                c_8      ⌊n/2⌋
9   else return j                          c_9      1
```

I began my analysis by trying to determine the time function of both partitions and I thought I would attach a picture of my work. I soon realized this wouldn't yield any results.