1 Problem Definition

Binary Search is an efficient algorithm for finding a target value within a sorted array. It reduces the problem size by half at each step, making it more efficient than a linear search for large arrays.

1.1 Inputs

- A sorted array of integers, A.
- A target integer, target, to find within A.

1.2 Outputs

- The index of target in the array A if target is found.
- -1 if target is not found in A.

1.3 Relationship

- The algorithm checks the middle of the array; if the middle value is the target, the search is over.
- If the middle value is greater than the target, the search continues in the left half of the array.
- If the middle value is less than the target, the search continues in the right half of the array.

2 Pseudocode for Binary Search Algorithms

2.1 Non-Recursive Binary Search[1]

```
Algorithm NonRecursiveBinarySearch(A, target)
    Input: An array A of n elements sorted in ascending order,
           and a target value target
    Output: The index of target in A or -1 if target is not found

    low <- 0
    high <- length(A) - 1

    while low <= high do
        mid <- (low + high) / 2
        if A[mid] = target then
            return mid
        else if A[mid] < target then
            low <- mid + 1
        else
            high <- mid - 1

    return -1
```

2.2 Recursive Binary Search

```
Algorithm RecursiveBinarySearch(A, low, high, target)
    Input: An array A, low and high indices defining the current subarray,
           and a target value target
    Output: The index of target in A or -1 if target is not found

    if high < low then
        return -1

    mid <- (low + high) / 2

    if A[mid] = target then
        return mid
    else if A[mid] < target then
        return RecursiveBinarySearch(A, mid + 1, high, target)
    else
        return RecursiveBinarySearch(A, low, mid - 1, target)
```

3 Analysis of Non-Recursive Binary Search [1]

The non-recursive binary search algorithm involves repeatedly dividing the search interval in half. If the interval is empty, the algorithm stops and returns -1. Each iteration of the while loop halves the search space.

3.1 Time Complexity Function $T(n)$

- At the first step, the array size is n.
- At the second step, it is $n/2$.
- At the third step, it is $n/4$, and so forth.

This continues until the size is reduced to 1. The number of iterations k needed until the array is reduced to size 1 can be determined by $n/2^k = 1$, leading to $k = \log_2(n)$. Therefore, the time complexity is $T(n) = O(\log n)$.

3.2 Recursion Equation

$$T(n) = T(n/2) + c$$

where c is the constant time to perform the mid calculation and comparison.

3.3 Recursion Tree

- The first level contributes c.
- The second level contributes $c/2$.
- The third level contributes $c/4$, and so forth.

The sum of contributions up to infinity is a converging geometric series:

$$T(n) = c + \frac{c}{2} + \frac{c}{4} + \cdots = 2c$$

3.3.1 Master Method[2]

We can use the Master Theorem to find the time complexity. Let $T(n)$ be the number of comparisons we perform in the worst case if the input array has α elements. Since we halve the active part and do at most two comparisons in each iteration, the recurrence is:

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

The theorem uses the generic form:

$$T(n) = \alpha T(\frac{n}{b}) + f(n)$$

and compares $f(n)$ to $n^{log_b a}$. In our case, $\alpha = 1$ and $\beta = 2$, so $n^{log_b a} = n^{log_b 1} = n^0 = 1$ and $f(n) = 2 \exists \mathcal{O}(1)$.

Therefore, it holds that:

$$2 = f(n) \exists \mathcal{O}(1) = \mathcal{O}(n^{log_b 1})$$

.

From the Master Theorem, we get the following:

$$T(n) \exists \mathcal{O}(n^{log_b a} logn) = \mathcal{O}(n^0 logn) = \mathcal{O}(logn)$$

4 Algorithm Implementation

4.1 Insertion Sort

The Insertion Sort algorithm is implemented in Python as follows:

```
    def insertion_sort(
        a:list ,
        n:int ,
    )->list:
        global ops_insert
        for i in range(1,n):
            k = a[i]
            j = i - 1
```

```python
            while j >= 0 and a[j] > k:
                a[j + 1] = a[j]
                ops_insert += 1
                j = j - 1
            a[j + 1] = k
        return a
```

For insertion sort, I chose to use a counter indicator inside the while loop as that is the scope of where the taxing operation of moving the values inplace takes place before the insertion.

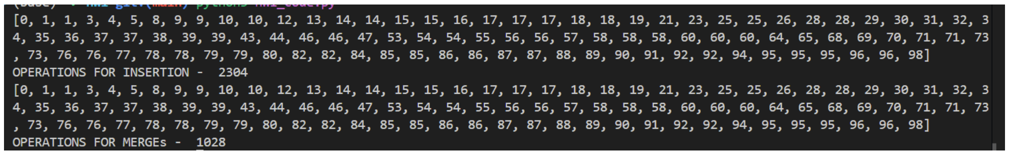
4.2 Merge Sort

The Merge Sort algorithm is implemented with a focus on comparing and merging elements:

```python
    def merge(
        a:list,
        p:int,
        q:int,
        r:int,
    )->None:
        global ops_merge
        nl = q - p + 1  # Number of elements in left subarray
        nr = r - q      # Number of elements in right subarray
        left = [0] * nl
        right = [0] * nr

        for i in range(nl):
            left[i] = a[p + i]
            ops_merge += 1
        for j in range(nr):
            right[j] = a[q + j + 1]
            ops_merge += 1

        i = j = 0
        k = p
        while i < nl and j < nr:
            ops_merge += 1
            if left[i] <= right[j]:
                a[k] = left[i]
                i += 1
            else:
                a[k] = right[j]
                j += 1
            k += 1

        # Copy remaining elements of left, if any
        while i < nl:
            a[k] = left[i]
            i += 1
            k += 1

        # Copy remaining elements of right, if any
        while j < nr:
            a[k] = right[j]
            j += 1
            k += 1

    def merge_sort(
        a:list ,
        p:int ,
        r:int ,
    )->list :
        if p >= r:
            return
        q = (p + r) // 2
        merge_sort(a, p, q)
        merge_sort(a, q + 1, r)
        merge(a, p, q, r)
        return a
```

For merge sort, I put a counter operation in the assigning of values to the left and right sub array loops and in the while loop where the core merging of left and right sub arrays takes place. The remaining loops in the merge function is non as consequential to the core merging algorithm as they just assign back to the array whatever is left.

4.3 Implementation Output



```
[0, 1, 1, 3, 4, 5, 8, 9, 9, 10, 10, 12, 13, 14, 14, 15, 15, 16, 17, 17, 17, 18, 18, 19, 21, 23, 25, 25, 26, 28, 28, 29, 30, 31, 32, 3
4, 35, 36, 37, 37, 38, 39, 39, 43, 44, 46, 46, 47, 53, 54, 54, 55, 56, 56, 57, 58, 58, 58, 60, 60, 60, 64, 65, 68, 69, 70, 71, 71, 73
, 73, 76, 76, 77, 78, 78, 79, 79, 80, 82, 82, 84, 85, 85, 86, 86, 87, 87, 88, 89, 90, 91, 92, 92, 94, 95, 95, 95, 96, 96, 98]
OPERATIONS FOR INSERTION -  2304
[0, 1, 1, 3, 4, 5, 8, 9, 9, 10, 10, 12, 13, 14, 14, 15, 15, 16, 17, 17, 17, 18, 18, 19, 21, 23, 25, 25, 26, 28, 28, 29, 30, 31, 32, 3
4, 35, 36, 37, 37, 38, 39, 39, 43, 44, 46, 46, 47, 53, 54, 54, 55, 56, 56, 57, 58, 58, 58, 60, 60, 60, 64, 65, 68, 69, 70, 71, 71, 73
, 73, 76, 76, 77, 78, 78, 79, 79, 80, 82, 82, 84, 85, 85, 86, 86, 87, 87, 88, 89, 90, 91, 92, 92, 94, 95, 95, 95, 96, 96, 98]
OPERATIONS FOR MERGEs -  1028
```

References

[1] ChatGPT. *Personal communication on binary search algorithms*. Conversation with an AI developed by OpenAI, conducted on September 1, 2023. 2023.

[2] *The Complexity of Binary Search*. `https://www.baeldung.com/cs/binary-search-complexity`. Accessed: 2024-08-31.