Homework Assignment: Algorithm Analysis and Implementation

Mauricio Estrella

September 12, 2025

1 Analyzing Problem

1.1 Analyzing Binary Search Algorithms

1.1.1 Problem Definition

Accurately define the problem by specifying the valid inputs and outputs, as well as their relationship.

- **Input:**
  - A sequence of sorted n numbers stored in array A[1:n] and a key value x.
- **Output:**
  - The index of the key x if it is present in the array A.
  - A special value (e.g., -1 or null) if the key x is not found in the array.
- **Relationship:** The output depends on whether the key x exists in the sorted input array A. The algorithm leverages the sorted property of the array to efficiently locate the key by repeatedly dividing the search interval in half.

1.1.2 Pseudocode

Write the pseudocode for two binary search algorithms, one recursive and the other non-recursive.

Algorithm 1 Binary Search

Function BINARYSEACH(A, x)

1: $low = 1$
2: $high = A.length$
3: **while** $low \leq high$ **do**
4: $mid = floor((low + high)/2)$
5: **if** $A[mid] == x$ **then**
6: **return** mid
7: **else if** $A[mid] < x$ **then**
8: $low = mid + 1$
9: **else**
10: $high = mid - 1$
11: **return** NOTFOUND

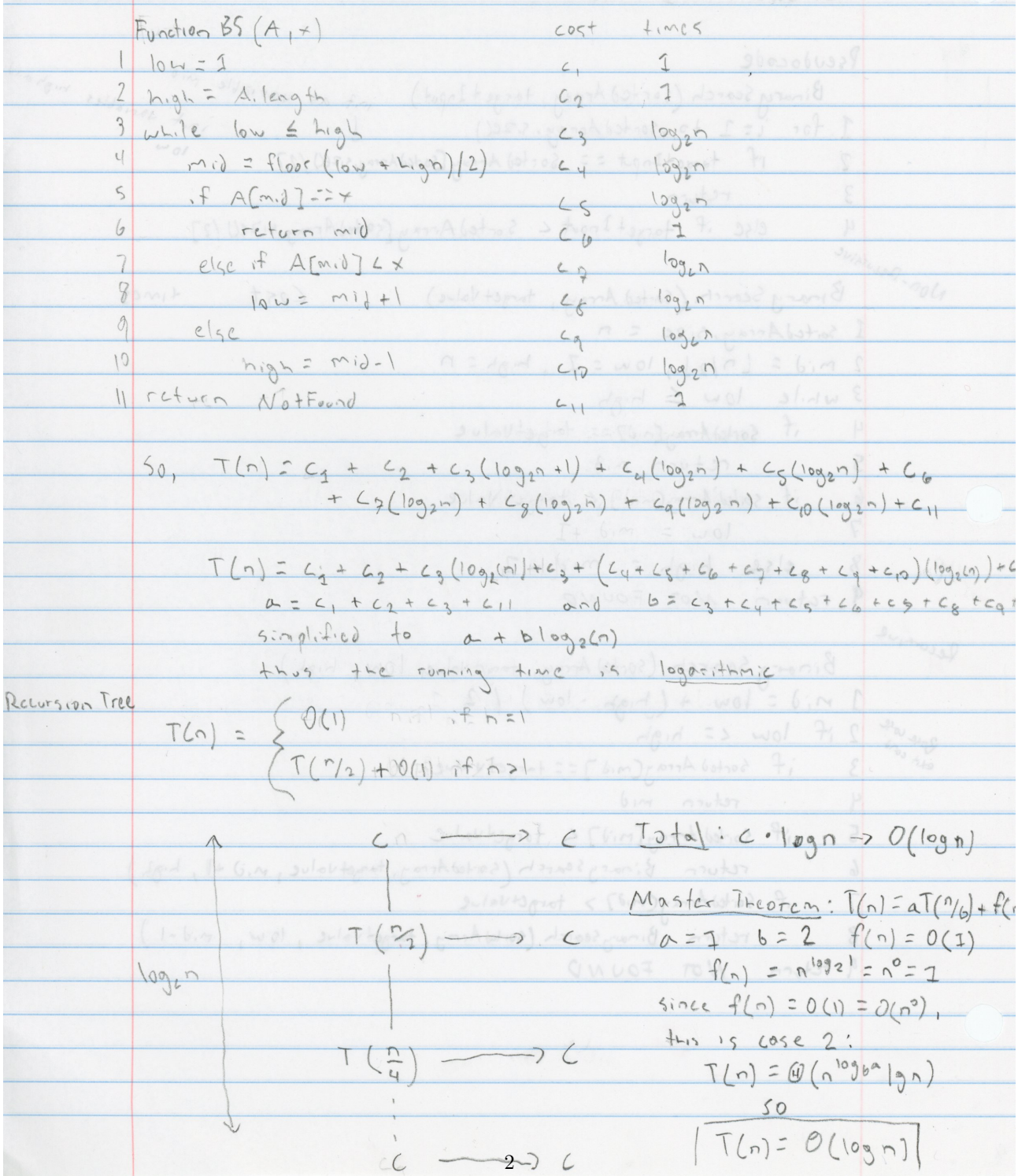
Algorithm 2 Recursive Binary Search

Function BINARYSEARCH(A, x, low, $high$)

1: **if** $low > high$ **then**
2: **return** NOTFOUND
3: **if** $low \leq high$ **then**
4: $mid = floor((low + high)/2)$
5: **if** $A[mid] == x$ **then**
6: **return** mid
7: **else if** $A[mid] < x$ **then**
8: BINARYSEARCH(A, x, $mid + 1$, $high$)
9: **else**
10: BINARYSEARCH(A, x, low, $mid - 1$)

1.1.3 Non-Recursive and Recursive Time Function Analysis

Decide the time function T(n) of the non-recursive algorithm, similar to how Insertion Sort is analyzed.



2 Programming Problem

2.1 Task 1 - Explanation of Indicators

Explain why the counters you chose are appropriate for measuring the time complexity of the sorting algorithms.

I chose to use two counters to indicate a proper measurement of time cost. The two counters initialized are to keep track of comparisons and assignments/swaps. These two tasks are fundamental in our algorithms and they represent the repeating tasks the CPU has to do. Counting these will give us a much better, machine-independent measure of the algorithm's cost than just using a stopwatch.

2.2 Task 2 - Test Cases and Sample Runs

Use random number generator to generate testing cases and use them on both algorithms, and use sample runs to show that the program works properly. Decide the size and number of testing cases so that there are enough data to show the difference between the two algorithms on time complexity, and at the same time the test does not run too long.

Sample runs:

```
--- Starting Sample Run for Insertion Sort ---
Original list: [23, 3, 10, 2]
Sorted list: [2, 3, 10, 23]
Total comparisons: 6
Total assignments: 11
--- End of Sample Run ---
--- Starting Sample Run for Merge Sort ---
Original list: [23, 3, 10, 2, 55, 14]
Sorted list: [2, 3, 10, 14, 23, 55]
Total comparisons: 11
Total assignments: 16
--- End of Sample Run ---
```

For testing, I decided to use sizes of n = 100, 500, 1000, 2500, and 5000. I just these so that it can show the dramatic difference in performance between the two algorithms. Results shown in next section.

2.3 Task 3 - Results

Collect and display the testing results to show the difference of the two algorithms on time complexity.

Input Size	Algorithm	Comparisons	Assignments
100	Insertion Sort	2710	2811
100	Merge Sort	536	672
500	Insertion Sort	62276	62781
500	Merge Sort	3849	4488
1000	Insertion Sort	250402	251408
1000	Merge Sort	8714	9976
2500	Insertion Sort	1582181	1584684
2500	Merge Sort	25082	28404
5000	Insertion Sort	6366068	6371074
5000	Merge Sort	55226	61808

2.4 Task 4 - Comparing Results and Discussion

Compare the testing results with the theoretical analysis conclusions on the algorithms and discuss their difference, if any.

- **Theory:** Insertion Sort's average time complexity is $O(n^2)$ and Merge Sort's is $O(nlogn)$
- **Results:** As the results table shows, the amount of comparisons and assignments are significantly less in Merge Sort, supporting out theoretical analysis. Insertion Sort performed around 115 times more work than Merge Sort which differs slightly from the 407 times more work in theory. This means that while Merge Sort's growth rate is fundamentally better, our particular implementation of Insertion Sort was quite efficient for its type (a small c), and our Merge Sort had some overhead (a larger c). This difference is what accounts for the gap between the experimental and theoretical ratios.