Performance of Search Algorithms on *n sized* Array

Muhammad Shahwaiz Abbasi

VT 2026

Introduction

This report presents the third assignment for the Data Structures and Algorithms (ID1021) course.

We will search for a number in an array using different methods and compare their runtime. We will be experimenting with the following:

- Search in an **Unsorted** array
- Search in a **sorted** array
- Use **Binary Search**
- Use **Recursive binary search**

I measured time in nanoseconds using a function from `Assignment 1` and then converted it to milliseconds to make the graph clearer and more readable.

The objective of this assignment is to understand how searching in arrays works and measure how the execution time changes with increasing array size.

Unsorted Array

Our first task in this assignment is to execute the unsorted search function and see its time performance. We wrote our own main function where we create arrays of various sizes and filled with random numbers. The searched number was chosen by me. The array sizes used range from one thousand to one million. We then find our given number in the array using the unsorted search function. `Code 1` uses a **For** loop to check if the number at the current index is equal to number we asked for. If a match is found, it returns true, else a message will be displayed saying **Key .... not found in the array**

```
bool unsorted_search(int array[], int length, int key) {
    for (int index = 0; index < length; index++) {
        if (array[index] == key) {
            return true;
        }
    }
    return false;
}
```

Listing 1: Code 1: Unsorted Search Function

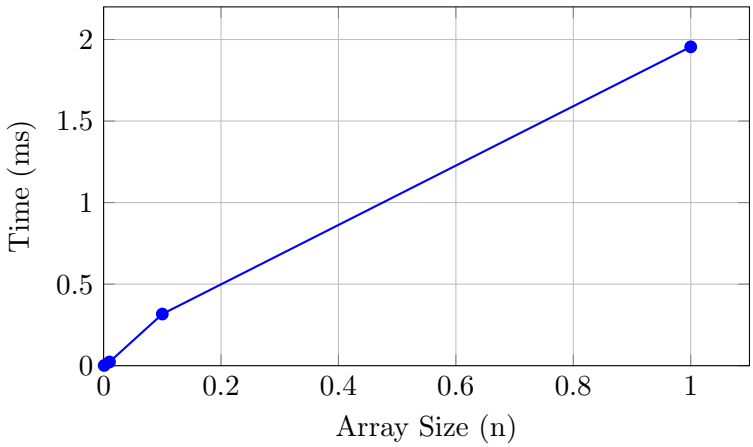

Figure 1: Unsorted Search algorithm time performance

As seen in `Figure 1`, when the array size increases, the runtime increases linearly. The reason behind this is that each element added to an array causes the processor to perform an additional comparison. So the time complexity for the search algorithm can be written as $\mathcal{O}(n)$.

Sorted Array

`Code 2` creates a sorted array, which we then use to search for a random number. Searching in a sorted array is faster than in an unsorted array because of early stopping but the runtime still grows linearly as the array size increases. This demonstrates that if we use a simple linear search, the time complexity remains $\mathcal{O}(n)$ regardless of whether the data is sorted or not.

The sorted search function checks if the numbers in the array are larger than the number searched. If true, then it returns false because the next number cannot be smaller, it has to be larger. This reduces runtime but

```
int *sorted(int n) {
    int *array = (int*)malloc(n*sizeof(int));
    int nxt = 0;
    for (int i = 0; i < n ; i++) {
        nxt += rand()%10 + 1;
        array[i] = nxt;
    }
    return array;
}
```

Listing 2: Code 2: Sorted Search Function

it will not change the time complexity. Therefore, both the sorted and unsorted search algorithms will have the same time complexity which can also be seen in `Figure 2`.

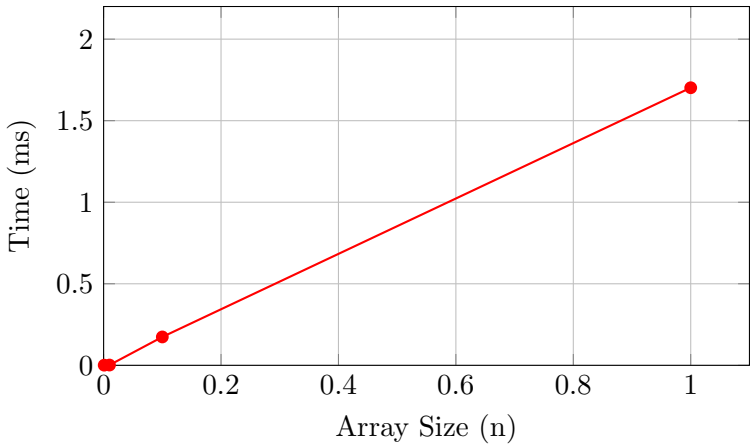

Figure 2: Sorted Search algorithm time performance

Comparison Between unsorted and sorted search algorithms

From the information above, we can conclude that the sorted search is faster than the unsorted search because it can stop early when the numbers become larger than the key. However, both algorithms still search linearly through the array. In the worst case, both methods must scan every element, so they have the same time complexity $\mathcal{O}(n)$.

Figure 3: Sorted vs Unsorted Search Performance

Binary Search

```
bool binary_search(int array[], int length, int key) {
    int first = 0;
    int last = length-1;
    while (true) {
        int index = (first + last) / 2; // jump to the middle
        if (array[index] == key) {
            return true; // hmm what now?
        }
        if (array[index] < key && index < last) {
            first = index + 1;  // what is the first possible
            ↪  page?
            continue;
        }
        if (array[index] > key && index > first) {
            last = index - 1; // what is the last possible
            ↪  page?
            continue;
        }
        return false;  // Why do we land here? What should we
        ↪  do?
```

Listing 3: Code 3: Binary Search Function

`Code 3` is a Binary Search algorithm function that takes a sorted array, the length of the array and the key to be found. We used `Code 2` to sort the

array. The binary algorithm starts by jumping to the middle of the array and checking if it matches the key, if it does, it returns true. If the integer in the middle is larger than the target key, the last index becomes the middle index - 1. Similarly, if the integer in the middle is smaller than the target key, the first index becomes the middle index + 1. I tested this code using large array sizes and as the size increases, the time increases only slightly. The assignment also required measuring the time for an array of 1 million elements, which took approximately 2000 ns while an array of 64 million elements took 3000 ns. The reason why the time only increases by 1000 ns is that the algorithm only has to perform about 6 extra comparisons because 1 million elements require 20 comparisons and 64 million elements require only 26. Last but not least, sorting is very helpful when there are many elements in an array. It reduces runtime because the number of comparisons is lower when using binary search.

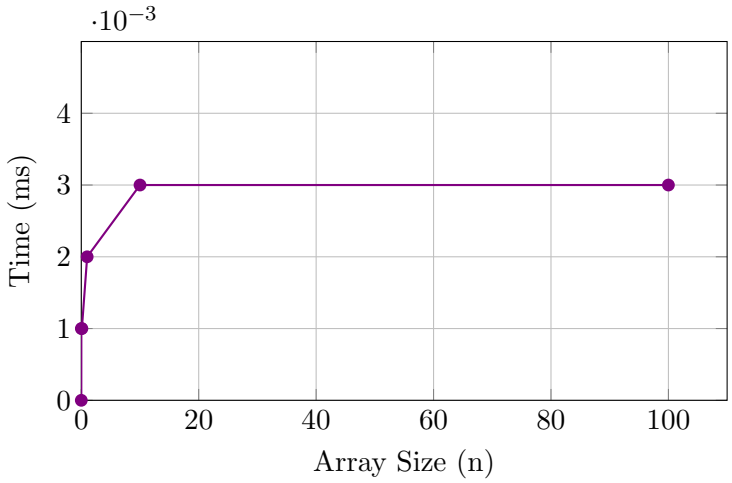

Figure 4: Binary Search algorithm time performance

Looking at `Figure 4`, the time increases quickly at the beginning when the array size increases but as the array size grows by $\boldsymbol{n}$, the time starts increasing more slowly, giving us a logarithmic curve on a graph. This represents a time complexity of $\mathcal{O}(\log n)$.

Recursive Binary Search

`Code 5` implements the recursive binary search algorithm. Instead of using a while loop, the function calls itself recursively to change the values of first and last. The graph in Figure 5 is similar to Figure 4 but there is a slight difference, the runtime is slightly higher than the standard binary search. However, the graph clearly shows a logarithmic function, confirming

that the time complexity remains $O(\log n)$.

Is the number of recursive calls on the stack a problem? The answer is no, because recursive binary search grows logarithmically, so for one million elements, it takes about 20 comparisons.

```
bool recursive(int array[], int length, int key, int first,
↪   int last) {
    int index = (first + last) / 2; // jump to the middle
    if (array[index] == key) {
        return true;
    }
    if (array[index] < key && index < last) {
    // call recursive but narrow the search
        return recursive(array, length, key, index+1, last);
    }
    if (array[index] > key && index > first) {
    // call recursive but narrow the search
        return recursive(array, length, key, first, index-1);
    }
        return false; // as before
}
```

Listing 4: Code 4: Recursive binary Search Function

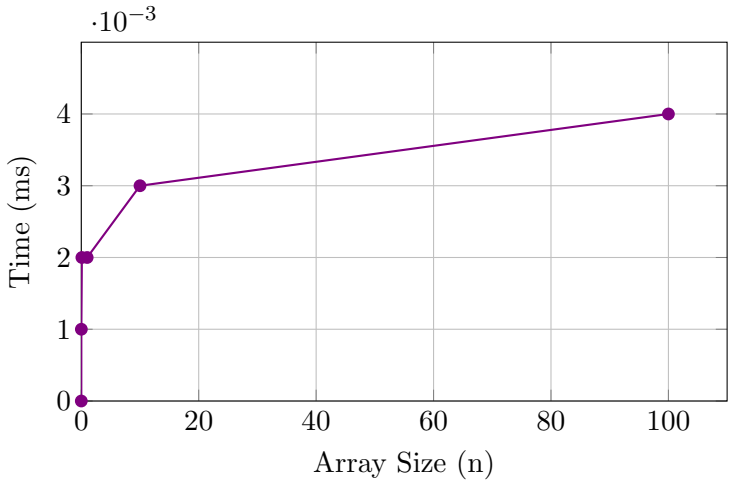

Figure 5: Recursive Binary Search algorithm time performance