# Project 1: Performance Measurement (Search)

March 2, 2025

# Contents

# 1 Chapter1: Introduction

As is well known, information retrieval is very important in today's computer networks. Today we are going to implement binary search and sequential search algorithms with both iterative and recursive versions. These algorithms will be used to search for N in an ordered list of N integers, from 0 to N-1. Thus we can figure out and compare the worst-case complexities and performance of these searching methods under different values of N.

## 1.1 My Task

Comparing the performance of two search algorithms - sequential search and binary search - in terms of their worst-case complexities and actual runtime performance while the goal is to understand how these algorithms behave under different input sizes.

## 1.2 What I will do

- Code both iterative and recursive implementations of the binary search and sequential search algorithms.
- Quantitatively assess and contrast the worst-case runtime of the developed algorithms across different input sizes, from N = 100 to N = 10,000. Then show the results in a table and a graph.
- Examine the worst-case complexity of each algorithm to gain insight into their performance as the input size grows.

## 1.3 Why I Do It

- Grasping the worst-case complexities of search algorithms aids in forecasting their efficiency as the size of the input data grows.
- Why I choose to compare the worst-case? Because the worst-case complexity provides an upper bound on the performance of an algorithm, which is useful for understanding how the algorithm behaves under different input sizes.

- Evaluating the efficiency of sequential versus binary search algorithms allows for informed choices about the most suitable algorithm, contingent on the scale and characteristics of the input data.

By doing this task, you'll obtain key insights into sequential and binary search algorithms, aiding in informed usage choices. So let's get started!

# 2 Chapter2: Algorithm Specification

## 2.1 Binary Search (Iterative)

**Algorithm 1:** Binary Search (Iterative)

**Input:** $arr$, $N$, $target$

**Output:** Index of $target$ in $arr$ or $-1$ if not found

$low \leftarrow 0$;

$high \leftarrow N - 1$;

**while** $low \leq high$ **do**

  $mid \leftarrow \lfloor \frac{low+high}{2} \rfloor$;

  **if** $arr[mid] = target$ **then**

    **return** $mid$;

  **else**

    **if** $arr[mid] < target$ **then**

      $low \leftarrow mid + 1$;

    **else**

      $high \leftarrow mid - 1$;

**return** $-1$;

### 2.2 Binary Search (Recursive)

**Algorithm 2:** Binary Search (Recursive)

**Input:** $arr$, $low$, $high$, $target$

**Output:** Index of $target$ in $arr$ or $-1$ if not found

**Function** `binarySearchRecursive`($arr$, $low$, $high$, $target$)**:**

- **if** $low \leq high$ **then**
  - $mid \leftarrow \lfloor \frac{low+high}{2} \rfloor$;
  - **if** $arr[mid] = target$ **then**
    - **return** $mid$;
  - **else**
    - **if** $arr[mid] < target$ **then**
      - **return** `binarySearchRecursive`($arr$, $mid + 1$, $high$, $target$);
    - **else**
      - **return** `binarySearchRecursive`($arr$, $low$, $mid - 1$, $target$);
- **return** $-1$;

### 2.3 Sequential Search (Iterative)

**Algorithm 3:** Sequential Search (Iterative)

**Input:** $arr$, $N$, $target$

**Output:** Index of $target$ in $arr$ or $-1$ if not found

**for** $i \leftarrow 0$ ***to*** $N - 1$ **do**

- **if** $arr[i] = target$ **then**
  - **return** $i$;

**return** $-1$;

### 2.4 Sequential Search (Recursive)

**Algorithm 4:** Sequential Search (Recursive)

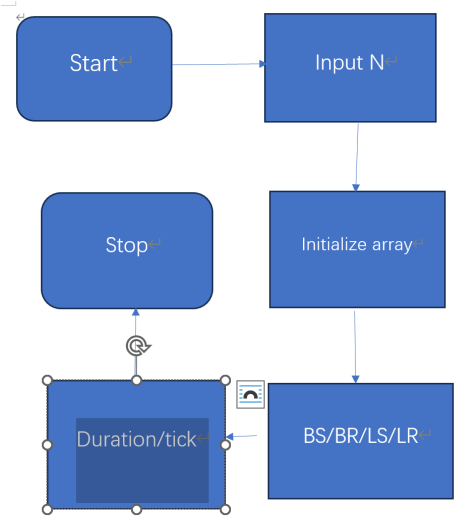
**Input:** $arr$, $N$, $target$, $index$

**Output:** Index of $target$ in $arr$ or $-1$ if not found

**Function** `sequentialSearchRecursive`($arr$, $N$, $target$, $index$)**:**

- **if** $index = N$ **then**
  - **return** $-1$;
- **if** $arr[index] = target$ **then**
  - **return** $index$;
- **else**
  - **return** `sequentialSearchRecursive`($arr$, $N$, $target$, $index + 1$);

Below is the sketch of main program




To calculate the total time and duration more accurately,I will repeat the function calls for K times(to decrease error,the suitable K is the one make the tick $> 50$).And to decrease the error,I will test each program 3 times and calculate the average.

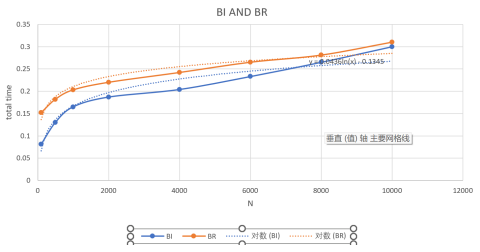
# 3 Chapter3: Testing Results

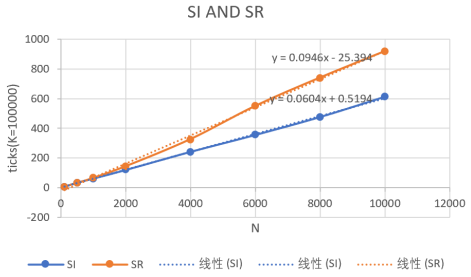
Below is the table of test data

| N | 100 | 500 | 1000 | 2000 | 4000 | 6000 | 8000 | 10000 |
|---|---|---|---|---|---|---|---|---|
| Binary Search (iterative version) | | | | | | | | |
| Iterations(K) | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 |
| Ticks | 81 | 130 | 165 | 187 | 204 | 233 | 265 | 300 |
| Total Time(sec) | 0.081 | 0.13 | 0.165 | 0.187 | 0.204 | 0.233 | 0.265 | 0.3 |
| duration(sec) | $8.1 \times 10^{-9}$ | $1.30 \times 10^{-8}$ | $1.65 \times 10^{-8}$ | $1.87 \times 10^{-8}$ | $2.04 \times 10^{-8}$ | $2.33 \times 10^{-8}$ | $2.65 \times 10^{-8}$ | $3.00 \times 10^{-8}$ |
| Binary Search (recursive version) | | | | | | | | |
| Iterations(K) | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 | 1000000 |
| Ticks | 152 | 182 | 203 | 220 | 242 | 265 | 281 | 310 |
| Total Time(sec) | 0.152 | 0.182 | 0.203 | 0.22 | 0.242 | 0.265 | 0.281 | 0.31 |
| duration(sec) | $1.52 \times 10^{-8}$ | $1.82 \times 10^{-8}$ | $2.03 \times 10^{-8}$ | $2.20 \times 10^{-8}$ | $2.42 \times 10^{-8}$ | $2.65 \times 10^{-8}$ | $2.81 \times 10^{-8}$ | $3.10 \times 10^{-8}$ |
| Sequential Search (iterative version) | | | | | | | | |
| Iterations(K) | 100000 | 100000 | 100000 | 100000 | 100000 | 100000 | 100000 | 100000 |
| Ticks | 75 | 342 | 62 | 120 | 242 | 356 | 477 | 614 |
| Total Time(sec) | 0.075 | 0.324 | 0.062 | 0.12 | 0.242 | 0.356 | 0.477 | 0.614 |
| duration(sec) | $7.5 \times 10^{-7}$ | $3.24 \times 10^{-6}$ | $6.2 \times 10^{-7}$ | $1.2 \times 10^{-6}$ | $2.42 \times 10^{-6}$ | $3.56 \times 10^{-6}$ | $4.77 \times 10^{-6}$ | $6.14 \times 10^{-6}$ |
| Sequential Search (recursive version) | | | | | | | | |
| Iterations(K) | 10000 | 10000 | 10000 | 10000 | 10000 | 10000 | 10000 | 10000 |
| Ticks | 53 | 303 | 652 | 144 | 327 | 551 | 742 | 925 |
| Total Time(sec) | 0.053 | 0.303 | 0.652 | 0.144 | 0.327 | 0.551 | 0.742 | 0.925 |
| duration(sec) | $5.3 \times 10^{-7}$ | $3.03 \times 10^{-6}$ | $6.52 \times 10^{-6}$ | $1.44 \times 10^{-5}$ | $3.27 \times 10^{-5}$ | $5.51 \times 10^{-5}$ | $7.42 \times 10^{-5}$ | $9.2 \times 10^{-5}$ |

# 4 Chapter4: Analysis and Comments

First, I would like to show the graph of the data table




The terms 'BI' represent iterative binary search, 'BR' represent recursive binary search, 'SI' represent iterative sequential search, and 'SR' represent recursive sequential search.

Next, we are going to analyze both the time and space complexities of all the algorithms, then compare them.

## 4.1 Sequential Search

### 4.1.1 Iterative version

1. Time Complexity: In the worst case, if the target element is located at the tail of the list or is absent altogether, the algorithm must traverse the entire list of n elements. Consequently, the time complexity is O(n), which can be reflected by the graph.

2. Space Complexity: The iterative version only uses a few variables for iteration, and the space used is constant regardless of the input size. Therefore, the space complexity is $O(1)$.

#### 4.1.2 Recursive version

1. Time Complexity: In the worst-case scenario, the recursive version may also need to traverse up to $n$ elements, leading to a time complexity of $O(n)$. This conclusion is reached by examining the recursion tree and counting the number of recursive invocations.

2. Space Complexity: With each recursive invocation, memory is allocated on the call stack. If the function must recurse $n$ times before it hits the base case, the space complexity will be $O(n)$, attributable to the $n$ layers of recursive calls that accumulate on the stack.

### 4.2 Binary Search

#### 4.2.1 Iterative version

1. Time Complexity: In the worst-case scenario, the time complexity of binary search is O(log n). This is because the algorithm divides the search interval in half with each step, leading to a logarithmic number of comparisons. The conclusion is also demonstrated from the graph,where the points form a logarithmic curve.

2. Space Complexity: The space complexity is O(1) for the iterative version, as it uses a constant amount of extra space.

#### 4.2.2 Recursive version

1. Time Complexity: Similar to the iterative version, the recursive version also halves the search space in each call. The time complexity is $O(\log n)$, derived from the depth of the recursion tree.

2. Space Complexity: O(log n) for the recursive version because the depth of call stack is log n.

### 4.3 Compare

#### 4.3.1 Recursive vs Iterative

- In the data table or graph, regardless of binary search or sequential search, the iterative version is faster than the recursive one. But their order of magnitudes and growth rate is the same.
- The reason why recursion is slower than loops may be that calling functions takes time.

#### 4.3.2 Binary search vs sequential search

- In normal cases where n is large, the binary algorithm is much faster. However, if the data is not ordered, the binary search will fail and sequential one can apply to all cases.

### 4.4 Conclusion

In conclusion, binary search stands out for reducing the number of comparisons to a minimum and is particularly appropriate for sorted arrays. However, the straightforwardness and efficiency of Sequential search for small data sets should not be dismissed. Each of these algorithms presents distinct trade-offs concerning time complexity and is appropriate for different situations depending on the characteristics and volume of the input data.

## 5 Appendix: Source code

Listing 1: search

```
1
2 //partial code
3 // Iterative binary search function
4 int iterativeBinary(int *array, int start, int end, int target) {
5     while (start <= end) { // Loop until start exceeds end
6         int mid = (start + end) / 2; // Find the middle index
7         if (target == array[mid]) {
8             return mid; // Target found, return index
```

```
9           } else if (target > array[mid]) {
10              start = mid + 1; // Search in the right half
11          } else {
12              end = mid - 1;   // Search in the left half
13          }
14      }
15      return -1; // Target not found
16  }
17
18  // Recursive binary search function
19  int recursiveBinary(int *array, int start, int end, int target) {
20      if (start <= end) { // Base case
21          int mid = (start + end) / 2; // Find the middle index
22          if (target == array[mid]) {
23              return mid; // Target found, return index
24          } else if (target > array[mid]) {
25              return recursiveBinary(array, mid + 1, end, target); // Searc
26          } else {
27              return recursiveBinary(array, start, mid - 1, target); // Sea
28          }
29      }
30      return -1; // Target not found
31  }
32
33  // Iterative sequential search function
34  int iterativeSequential(int *array, int size, int target) {
35      for (int i = 0; i < size; i++) { // Loop through the array
36          if (array[i] == target) {
37              return i; // Target found, return index
38          }
39      }
40      return -1; // Target not found
41  }
```

```
42
43 // Recursive sequential search function
44 int recursiveSequential(int *array, int start, int end, int target) {
45     if (start <= end) { // Base case
46         if (array[start] == target) {
47             return start; // Target found, return index
48         } else {
49             return recursiveSequential(array, start + 1, end, target); //
50         }
51     }
52     return -1; // Target not found
53 }
54
55 int main() {
56     // Measure time for recursive sequential search
57     begin = clock();
58     for (int i = 0; i < 1000; i++) { // Repeat for better accuracy
59         result = recursiveSequential(array, 0, s - 1, s); // Perform sear
60     }
61     stop = clock();
62     duration = ((double)(stop - begin)) / CLOCKS_PER_SEC; // Calculate du
63     printf("%lf seconds\n", duration);
64
65     return 0;
66 }
```

## 6 Declaration

I hereby declare that all the work done in this project titled "Project 1:Performance Measurement (Search) " is of my independent effort