Fundamentals of Data Structures

Projects 1: Performance Measurement (Search)

Author:

Date: 2025/03/16

2024-2025 Spring & Summer Semester

Table of Contents

Chapter 1: Introduction

For an ordered list of N integers, indexed from 0 to $N-1$, checking whether N is not in the list represents the worst-case scenario for both sequential search and binary search algorithms. This project aims to implement the following in C:

Tasks

a) Iterative and recursive implementations of sequential search and binary search.
b) Analysis of the worst-case time complexity of these four algorithms.
c) Execution and comparison of the performance of these four algorithms under the worst-case scenario with test data sizes of N

$$N = 100, 500, 1000, 2000, 4000, 6000, 8000, 10000$$

Chapter 2: Algorithm Specification

2.1) Time Measurement

To measure execution time, this project will use the `clock()` function from the C language `<time.h>` library. The structure of the time measurement code is as follows:

```
#include<stdio.h>
#include<time.h>

clock_t start, stop;
double duration;

int main(){
    start = clock();  //Record the start time
    Function();       //Call the function whose execution time is being measured
    stop = clock();   //Record the end time
    duration = ((double)(stop - start))/CLK_TCK;
    printf("stop = %ld, start = %ld, delta = %ld", duration, stop, start, stop -
start);
    return 0;
}
```

2.2) Iterative Binary Search

The **iterative binary search** is an efficient search algorithm that works on a sorted list. It repeatedly divides the search range in half until it finds the target value or determines that the value is not in the list, Its working **pseudocode** is shown below:

```
Function BinarySearch_Iterative(arr, left, right, target):
    While left <= right:
        mid = (left + right) / 2  // Calculate middle index

        If arr[mid] > target:
            right = mid - 1  // Search in the left half
        Else if arr[mid] < target:
            left = mid + 1  // Search in the right half
        Else:
            Return 1  // Target found

    Return 0  // Target not found
```

2.3) Recursive Binary Search

The **recursive binary search** is a divide-and-conquer algorithm used to efficiently search for a target value in a sorted list. **Instead of using a loop**, it calls itself recursively, reducing the search space by half at each step until the target is found or the search range becomes empty.Its working **pseudocode** is shown below:

```
Function BinarySearch_Iterative(arr, left, right, target):
    While left <= right:  // Continue searching while valid range exists
        mid = (left + right) / 2  // Find the middle index

        If arr[mid] > target:
            right = mid - 1  // Search in the left half

        Else if arr[mid] < target:
            left = mid + 1  // Search in the right half

        Else:
            Return 1  // Target found

    Return 0  // Target not found
```

2.4) Iterative Sequential Search

The **iterative sequential search** is a simple search algorithm that **scans a list element by element** until it finds the target value or reaches the end of the list. Its working **pseudocode** is shown below:

```
Function SequentialSearch_Iterative(arr, n, target):
    For i from 0 to n - 1:
        If arr[i] == target:
            Return 1  // Target found

    Return 0  // Target not found
```

2.5) Recursive Sequential Search

The **recursive sequential search** is a simple search algorithm that **checks each element of the list recursively** until it finds the target value or reaches the end of the list. Unlike the iterative approach, which uses a loop, **this method calls itself**

with a reduced problem size at each step. Its working **pseudocode** is shown below:

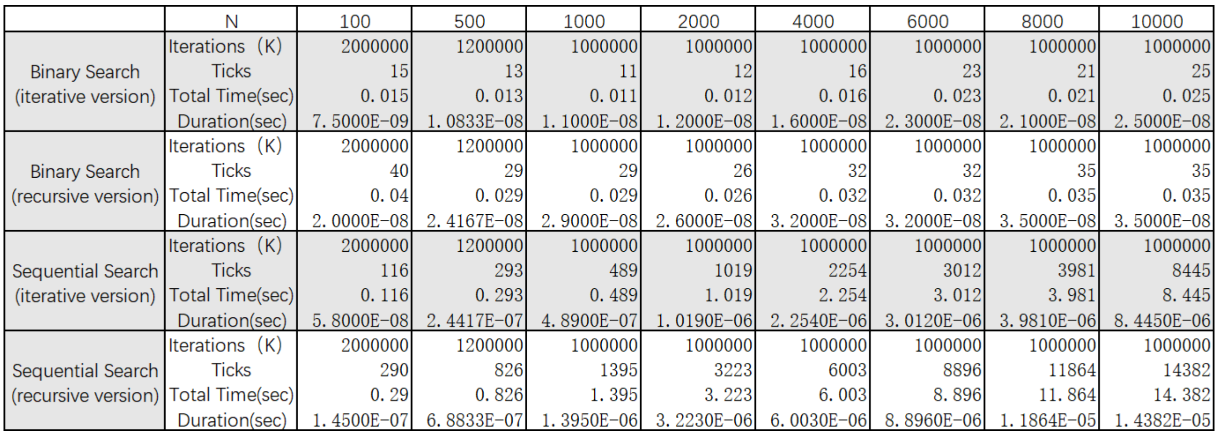
```
Function SequentialSearch_Recursive(arr, n, target):
    If n == 0:  // Base case: empty array
        Return 0  // Target not found

    If arr[n - 1] == target:
        Return 1  // Target found

    Return SequentialSearch_Recursive(arr, n - 1, target)  // Recursive call with
reduced size
```

Chapter 3: Testing Results

To ensure accurate measurement of execution time, if a function runs too quickly and takes less than a tick to complete, we repeat its execution K times. The total time is then divided by K to obtain the average duration for a single function call.

The repetition factor K must be chosen large enough so that at least 10 ticks elapse, ensuring a measurement accuracy of at least 10%.

The output data is listed below:



	N	100	500	1000	2000	4000	6000	8000	10000
Binary Search (iterative version)	Iterations （K）	2000000	1200000	1000000	1000000	1000000	1000000	1000000	1000000
	Ticks	15	13	11	12	16	23	21	25
	Total Time(sec)	0. 015	0. 013	0. 011	0. 012	0. 016	0. 023	0. 021	0. 025
	Duration(sec)	7. 5000E-09	1. 0833E-08	1. 1000E-08	1. 2000E-08	1. 6000E-08	2. 3000E-08	2. 1000E-08	2. 5000E-08
Binary Search (recursive version)	Iterations （K）	2000000	1200000	1000000	1000000	1000000	1000000	1000000	1000000
	Ticks	40	29	29	26	32	32	35	35
	Total Time(sec)	0. 04	0. 029	0. 029	0. 026	0. 032	0. 032	0. 035	0. 035
	Duration(sec)	2. 0000E-08	2. 4167E-08	2. 9000E-08	2. 6000E-08	3. 2000E-08	3. 2000E-08	3. 5000E-08	3. 5000E-08
Sequential Search (iterative version)	Iterations （K）	2000000	1200000	1000000	1000000	1000000	1000000	1000000	1000000
	Ticks	116	293	489	1019	2254	3012	3981	8445
	Total Time(sec)	0. 116	0. 293	0. 489	1. 019	2. 254	3. 012	3. 981	8. 445
	Duration(sec)	5. 8000E-08	2. 4417E-07	4. 8900E-07	1. 0190E-06	2. 2540E-06	3. 0120E-06	3. 9810E-06	8. 4450E-06
Sequential Search (recursive version)	Iterations （K）	2000000	1200000	1000000	1000000	1000000	1000000	1000000	1000000
	Ticks	290	826	1395	3223	6003	8896	11864	14382
	Total Time(sec)	0. 29	0. 826	1. 395	3. 223	6. 003	8. 896	11. 864	14. 382
	Duration(sec)	1. 4500E-07	6. 8833E-07	1. 3950E-06	3. 2230E-06	6. 0030E-06	8. 8960E-06	1. 1864E-05	1. 4382E-05

Figure 1: Output Data

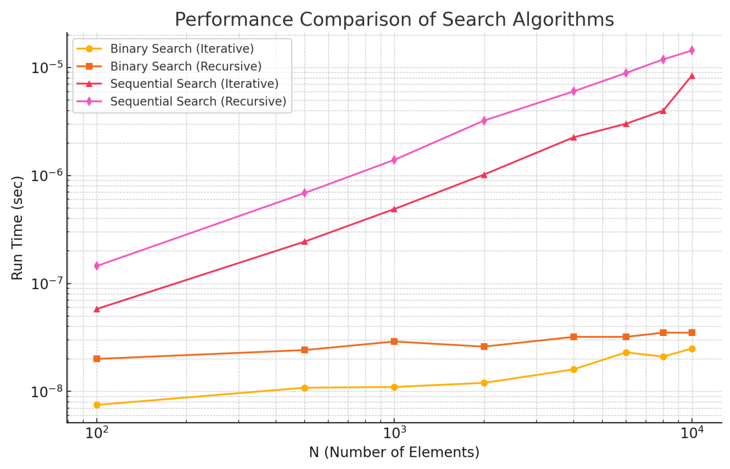


Figure 2: Performance Comparison Of Search Algorithms

Chapter 4: Analysis and Comments

4.1) Worst-Case Time Complexity Analysis

4.1.1) Binary Search (Iterative Version) - O(log N)

Binary search works by repeatedly dividing the sorted array into halves and checking whether the middle element is the target. In the worst case, the element is not found, requiring the search to continue until the array is reduced to a single element.

The recurrence relation for the number of steps is:

$$T(N) = T\left(\frac{N}{2}\right) + O(1)$$

Solving this recurrence using the recurrence tree method or the Master Theorem gives:

$$O(\log N)$$

This is highly efficient compared to sequential search, as the number of comparisons grows logarithmically with N.

4.1.2) Binary Search (Recursive Version) - O(log N)

The recursive version follows the same logic as the iterative version but uses function calls instead of a loop. In each recursive call, the search space is halved, leading to the recurrence relation:

$$T(N) = T\left(\frac{N}{2}\right) + O(1)$$

Again, solving this gives:

$$O(\log N)$$

The only difference from the iterative version is the extra space used by recursive function calls, which leads to an auxiliary space complexity of O(log N) due to function stack usage.

4.1.3) Sequential Search (Iterative Version) - O(N)

In the worst case, the target element is not in the array, so every element must be checked. The number of comparisons in the worst case is:

$$T(N) = O(N)$$

This is significantly worse than binary search, especially for large N, since performance degrades linearly as the input size grows.

4.1.4) Sequential Search (Recursive Version) - O(N)

Like the iterative version, the worst case requires checking every element. The recurrence relation for the recursive version is:

$$T(N) = T(N-1) + O(1)$$

Solving this gives:

$$T(N) = O(N)$$

The recursive implementation also suffers from additional function call overhead and stack space usage of O(N), making it inefficient.

The worst-case time complexities of the four searching methods are analyzed below.

Search Algorithm	**Time Complexity**
Binary Search (Iterative)	$O(\log N)$
Binary Search (Recursive)	$O(\log N)$
Sequential Search (Iterative)	$O(N)$
Sequential Search (Recursive)	$O(N)$

Binary search, both in its iterative and recursive versions, has a logarithmic time complexity of $O(\log N)$. This makes it significantly faster than sequential search, which operates in linear time, i.e., $O(N)$. However, the recursive version of binary search requires additional stack space, leading to an auxiliary space complexity of $O(\log N)$, whereas the iterative version has a space complexity of $O(1)$.

Sequential search, on the other hand, is inefficient for large datasets due to its $O(N)$ complexity. The recursive version is even less efficient, as it incurs additional function call overhead and uses $O(N)$ stack space.

4.2) Space Complexities Analysis

The following table provides a comparison of the space complexities:

Search Algorithm	**Space Complexity**
Binary Search (Iterative)	$O(1)$
Binary Search (Recursive)	$O(\log N)$
Sequential Search (Iterative)	$O(1)$
Sequential Search (Recursive)	$O(N)$

From the experimental results, it is evident that **binary search is far superior** to sequential search for sorted data. The **iterative version of binary search** is generally preferred due to its lower space requirements, while the **recursive version suffers from function call overhead**.

Summary

1. **Binary search is the optimal choice for sorted arrays** due to its $O(\log N)$ complexity.
2. **Iterative methods generally outperform recursive ones** due to lower memory overhead.
3. **Sequential search is inefficient for large datasets**, and alternative methods like hashing or interpolation search may be preferable.

Appendix: Source Code (in C)

File Proj_Search.c:

```c
#include <stdio.h>
#include <time.h>

clock_t start, stop;  // Variables to store the start and stop times
double duration;  // Stores the execution time

// Function prototypes
int BinaSearch_Circulation(int a[], int left, int right, int x);
int BinaSearch_Recursion(int a[], int left, int right, int x);
int SequeSearch_Circulation(int a[], int n, int x);
int SequeSearch_Recursion(int a[], int n, int x);

int main() {
    int n = 0, x = 0;
    int k = 1000000, status = -1;  // k is the number of test iterations

    // Read the input values: array size (n), target value (x), and the number of
```

```
iterations (k)
    scanf("%d %d %d", &n, &x, &k);
    int a[n];  // Declare an array of size n

    // Initialize the array with sequential values (0, 1, 2, ..., n-1)
    for (int i = 1; i < n; i++) {
        a[i] = i;
    }

    printf("Circulation Times k = %d\n", k);

    // Binary Search using Recursion
    printf("Binary Search by Recursion:\n");
    start = clock();  // Record start time
    for (int i = 0; i < k; i++) {
        status = BinaSearch_Recursion(a, 0, n, x);
    }
    stop = clock();  // Record stop time
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("Time is %f\n stop = %ld, start = %ld, delta = %ld\n\n", duration,
stop, start, stop - start);

    // Sequential Search using Recursion
    printf("Sequence Search by Recursion:\n");
    start = clock();
    for (int i = 0; i < k; i++) {
        status = SequeSearch_Recursion(a, n, x);
    }
    stop = clock();
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("Time is %f\n stop = %ld, start = %ld, delta = %ld\n\n", duration,
stop, start, stop - start);

    // Binary Search using Iteration
    printf("Binary Search by Circulation:\n");
    start = clock();
    for (int i = 0; i < k; i++) {
        status = BinaSearch_Circulation(a, 0, n, x);
    }
    stop = clock();
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("Time is %f\n stop = %ld, start = %ld, delta = %ld\n\n", duration,
stop, start, stop - start);

    // Sequential Search using Iteration
    printf("Sequence Search by Circulation:\n");
    start = clock();
    for (int i = 0; i < k; i++) {
        status = SequeSearch_Circulation(a, n, x);
    }
    stop = clock();
```

```c
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("Time is %f\n stop = %ld, start = %ld, delta = %ld\n\n", duration,
stop, start, stop - start);

    // Final result check
    if (status == 1) {
        printf("Find!\n");  // If status is 1, the target was found
    } else if (status == 0) {
        printf("Not Find!\n");  // If status is 0, the target was not found
    } else {
        printf("ERROR!\n");  // If status is neither 1 nor 0, there is an error
    }

    return 0;
}

// Iterative Binary Search (Loop-Based)
int BinaSearch_Circulation(int a[], int left, int right, int x) {
    int mid;
    while (left < right) {  // Continue searching while left is less than right
        mid = (left + right) / 2;  // Find the middle index
        if (a[mid] > x) {
            right = mid - 1;  // Search in the left half
        } else if (a[mid] < x) {
            left = mid + 1;  // Search in the right half
        } else {
            return 1;  // Target found
        }
    }
    return 0;  // Target not found
}

// Recursive Binary Search
int BinaSearch_Recursion(int a[], int left, int right, int x) {
    int mid = (left + right) / 2;  // Calculate the middle index
    if (left > right) {  // Base case: search range is invalid
        return 0;  // Target not found
    }
    if (a[mid] == x) {
        return 1;  // Target found
    } else if (a[mid] < x) {
        return BinaSearch_Recursion(a, mid + 1, right, x);  // Search in the right
half
    } else {
        return BinaSearch_Recursion(a, left, mid - 1, x);  // Search in the left
half
    }
}

// Iterative Sequential Search (Loop-Based)
int SequeSearch_Circulation(int a[], int n, int x) {
    for (int i = 0; i < n; i++) {  // Traverse the entire array
        if (x == a[i]) {  // If the current element matches the target
            return 1;  // Target found
        }
    }
    return 0;  // Target not found
}

// Recursive Sequential Search

int SequeSearch_Recursion(int a[], int n, int x){
    if(n == 0){ // Base case: if only one element left
        return 0;
    }
    if(a[n-1] == x){// Check if the  element matches the target
        return 1;
    }
    return SequeSearch_Recursion(a, n-1, x); // Recursively check remaining
elements

}
```

Declaration

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