Project 1 Performance Measurement (Search)

March 9, 2024

Contents

1 Chapter 1

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.

- Our 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.
- What to be done:
  1. Implement iterative and recursive versions of both binary search and sequential search algorithms.
  2. Analyze the worst-case complexities of each algorithm to understand their behavior with increasing input size.
  3. Measure and compare the worst-case performance of the implemented algorithms for varying input sizes ranging from N = 100 to N = 10000.
- Why we do it:
  1. Understanding the worst-case complexities of search algorithms helps in predicting their performance as the input size increases.
  2. Practical performance measurements provide empirical evidence of how these algorithms behave in real-world scenarios, complementing theoretical analysis.
  3. By comparing the performance of sequential search and binary search algorithms, one can make informed decisions about which algorithm to use based on the size and nature of the input data.

Thus,by completing this task, you'll gain valuable insights into the theoretical and practical aspects of sequential and binary search algorithms, enabling you to make informed decisions about their usage in different scenarios.

2 Chapter 2

- iterative version of Binarysearch:

Algorithm 1: Binary Search (Iterative)

Function `binarySearchIterative`($arr, N, target$):
    $low \leftarrow 0$;
    $high \leftarrow N - 1$;
    while $low \leq high$ **do**
        $mid \leftarrow \left\lfloor \frac{low+high}{2} \right\rfloor$;
        if $arr[mid] = target$ **then**
            return mid;
        else if $arr[mid] < target$ **then**
            $low \leftarrow mid + 1$;
        else
            $high \leftarrow mid - 1$;
    return -1;

- recursive version of Binarysearch

Algorithm 2: Binary Search (Recursive)

```
Function binarySearchRecursive(arr, low, high, target):
    if low <= high then
        mid ← ⌊(low+high)/2⌋;
        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;
```

- iterative version of Sequentialsearch

Algorithm 3: Sequential Search (Iterative)

```
Function sequentialSearchIterative(arr, N, target):
    for i ← 0 to N − 1 do
        if arr[i] = target then
            return i;
    return −1;
```

- recursive version of Sequentialsearch

Algorithm 4: Sequential Search (Recursive)

```
Function sequentialSearchRecursive(arr, N, target, index):
    if index < N then
        if arr[index] = target then
            return index;
        else
            return sequentialSearchRecursive(arr, N, target, index + 1);
    return −1;
```

Below is the sketch of the main program:

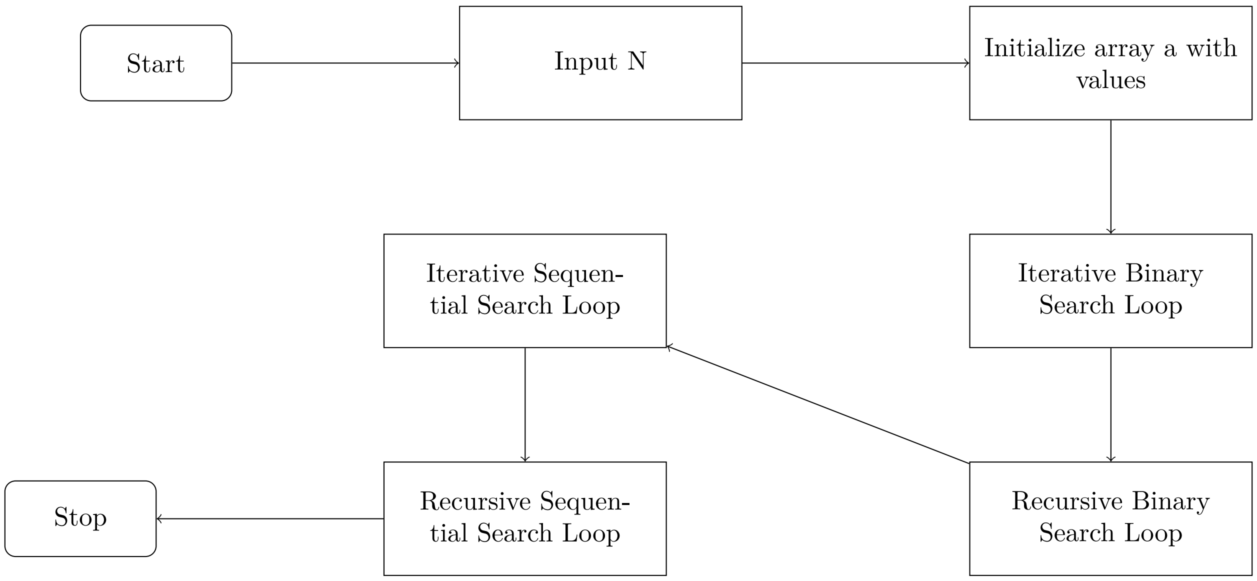

Figure 1: sketch of the program

3 Chapter 3

Below is the table of the data

	N	100	500	1000	2000	4000	6000	8000	10000
Binary Search (iterative version)	Iterations(K)	10000000	10000000	1000000	1000000	1000000	1000000	1000000	1000000
	Ticks	79	99	12	15	17	18	20	22
	Total Time(sec)	0.079	0.099	0.012	0.015	0.017	0.018	0.020	0.022
	Duration(sec)	7.9×10^{-9}	9.9×10^{-9}	1.2×10^{-8}	1.5×10^{-8}	1.7×10^{-8}	1.8×10^{-8}	2.0×10^{-8}	2.2×10^{-8}
Binary Search (recursive version)	Iterations(K)	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
	Ticks	11	14	16	17	19	20	21	22
	Total Time(sec)	0.011	0.014	0.016	0.017	0.019	0.020	0.021	0.022
	Duration(sec)	1.1×10^{-8}	1.4×10^{-8}	1.6×10^{-8}	1.7×10^{-8}	1.9×10^{-8}	$2.0 times 10^{-8}$	2.1×10^{-8}	2.2×10^{-8}
Sequential Search (iterative version)	Iterations(K)	1000000	100000	100000	100000	10000	10000	10000	10000
	Ticks	53	24	46	91	19	28	37	46
	Total Time(sec)	0.053	0.024	0.046	0.091	0.019	0.028	0.037	0.046
	Duration(sec)	5.3×10^{-8}	2.4×10^{-7}	24.6×10^{-7}	9.1×10^{-7}	1.9×10^{-6}	2.8×10^{-6}	3.7×10^{-6}	4.6×10^{-6}
Sequential Search (iterative version)	Iterations(K)	100000	100000	10000	10000	10000	10000	1000	1000
	Ticks	16	78	16	33	66	98	12	17
	Total Time(sec)	0.016	0.078	0.016	0.033	0.066	0.098	0.012	0.017
	Duration(sec)	1.6×10^{-7}	7.8×10^{-7}	1.6×10^{-6}	3.3×10^{-6}	6.6×10^{-6}	9.8×10^{-6}	1.2×10^{-5}	1.7×10^{-5}

4 Chapter 4

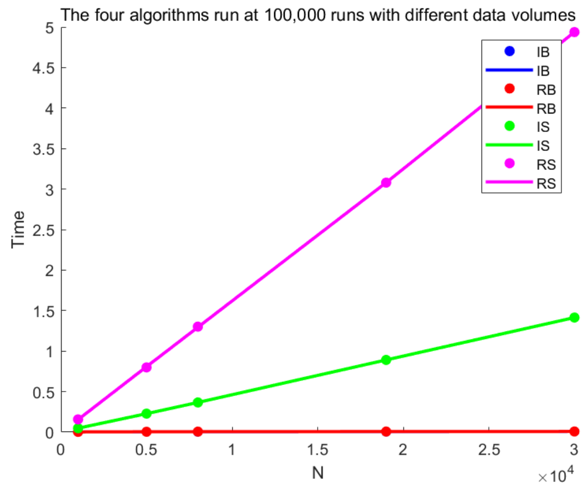

Figure 2: The four algorithms run at 100,000 runs with different data volumes

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

1. Sequential Search
   (a) Iterative version
      i. Time Complexity: In the worst-case scenario, when the target element is at the end of the list or not present, the algorithm needs to iterate through all n elements. Thus, the time complexity is O(n). This can be derived by analyzing the loop that iterates through the list.
      ii. 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).
   (b) Recursive version
      i. Time Complexity: The recursive version also potentially needs to iterate through all n elements in the worst case, resulting in a time complexity of O(n). This can be derived by analyzing the recursion tree and the number of recursive calls made.
      ii. Space Complexity: Each recursive call consumes memory on the call stack. In the worst case, if the function recurses n times before reaching the base case, the space complexity is O(n) due to the n levels of recursion on the call stack.
2. Binary Search
   (a) Iterative version
      i. Time Complexity: In each iteration, the search space is halved. The time complexity can be derived by solving the equation $n/2^k = 1$, where k is the number of iterations. By solving this, we get $k = \log n$, leading to a time complexity of $\mathrm{O}(\log n)$.
      ii. 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).
   (b) Recursive version

[0]The terms 'IB' represent iterative binary search, 'RB' represent recursive binary search, 'IS' represent iterative sequential search, and 'RS' represent recursive sequential search.

i. Time Complexity: Similar to the iterative version, the time complexity is $O(\log n)$ as the search space is halved in each recursive call. This can be derived by analyzing the recursion tree and the number of recursive calls made.

ii. Space Complexity:During each recursive call, the space is utilized for maintaining the function call stack, which includes parameters, return addresses, and local variables. Since the binary search algorithm divides the array into smaller subarrays, the maximum depth of the call stack is $O(\log n)$, where n is the size of the input array.Taking into account both the input space and the auxiliary space, the total space complexity of the recursive binary search algorithm is $O(\log n)$, where n represents the size of the input array.

Comparations are listed as follows:

Binary search offers a time complexity of O(log n) due to its ability to halve the search space with each comparison. This makes it significantly more efficient than Sequential search, especially for large input sizes. On the other hand, Sequential search has a time complexity of O(n) in the worst case, as it may need to iterate through every element in the input array.

Furthermore, both binary search and Sequential search have a space complexity of O(1), meaning they require a constant amount of additional space regardless of the input size. This demonstrates their efficiency in terms of memory usage.

In conclusion, while binary search excels in minimizing the number of comparisons and is well-suited for sorted arrays, Sequential search's simplicity and effectiveness for small datasets cannot be overlooked. Both algorithms offer different trade-offs in terms of time complexity and are suitable for different scenarios based on the nature and size of the input data.

5 Source code

```c
#include <stdio.h>  // Include standard input and output library
#include <time.h>   // Include time library for time-related functions

clock_t begin, stop; // Declare variables to record time
double duration;     // To calculate the running time

// Function for iterative binary search
int iterativeBinary(int *a, int start, int end, int N) {
    while (start < end) { // Continue loop until start and end meet
        int mid = (start + end) / 2; // Calculate the middle index
        if (N == a[mid]) {
            return mid; // If N is found at mid, return the index
        } else if (N > a[mid]) {
            start = mid + 1; // Update start for right half
        } else {
            end = mid - 1;   // Update end for left half
        }
    }
    return -1; // Return -1 if element is not found
}

// Function for recursive binary search
int recursiveBinary(int *a, int start, int end, int N) {
    if (start < end) { // Base case: when start exceeds end
        int mid = (start + end) / 2; // Calculate the middle index
        if (N == a[mid]) {
            return mid; // If N is found at mid, return the index
```

```
        } else if (N > a[mid]) {
            return recursiveBinary(a, mid + 1, end, N); // Search the right half
        } else {
            return recursiveBinary(a, start, mid - 1, N); // Search the left half
        }
    }
    return -1; // Return -1 if element is not found
}

// Function for iterative sequential search
int iterativeSequential(int *a, int end, int N) {
    for (int i = 0; i <= end; i++) { // Iterate through the array
        if (a[i] == N) {
            return i; // If N is found, return the index
            break;     // Break the loop after finding the element
        }
    }
    return -1; // Return -1 if element is not found
}

// Function for recursive sequential search
int recursiveSequential(int *a, int start, int end, int N) {
    if (start <= end) { // Base case: when start exceeds end
        if (a[start] == N) {
            return start; // If N is found at index start, return the index
        } else {
            return recursiveSequential(a, start + 1, end, N); // Recur for the rest of the array
        }
    }
    return -1; // Return -1 if element is not found
}
int main() {
    int N;
    scanf("%d", &N); // Input the size of the array

    int a[N], m;
    for (int i = 0; i < N; i++) {
        a[i] = i; // Populate the array with values from 0 to N-1
    }

    // Measure the time taken for each search algorithm
    begin = clock(); // Record the starting time
    for (int i = 0; i < 100000; i++) { // Repeat 100000 times for better time measurement
        m = iterativeBinary(a, 0, N - 1, N); // Perform iterative binary search
    }
    stop = clock(); // Record the stopping time
    duration = ((double)(stop - begin)) / CLOCKS_PER_SEC; // Calculate the duration in seconds
    printf("Tick for iterativeBinary: %lf, Time for iterativeBinary: %lf\n", (double)(stop - begin), duration);

    begin = clock(); // Record the starting time
```

```
    for (int i = 0; i < 100000; i++) { // Repeat 100000 times for better time measurement
        m = recursiveBinary(a, 0, N - 1, N); // Perform recursive binary search
    }
    stop = clock(); // Record the stopping time
    duration = ((double)(stop - begin)) / CLOCKS_PER_SEC; // Calculate the duration in seconds
    printf("Tick for recursiveBinary: %lf, Time for recursiveBinary: %lf\n", (double)(stop - begin), duration);

    begin = clock(); // Record the starting time
    for (int i = 0; i < 100000; i++) { // Repeat 100000 times for better time measurement
        m = iterativeSequential(a, N - 1, N); // Perform iterative sequential search
    }
    stop = clock(); // Record the stopping time
    duration = ((double)(stop - begin)) / CLOCKS_PER_SEC; // Calculate the duration in seconds
     printf("Tick for iterativeSequential:%lf,Time for iterativesequential:%lf\n",(double)(stop-begin),duration);\
    begin=clock();
    for (int i = 0; i < 100000; i++)
    {
        m=recursiveSequential(a,0,N-1,N);
    }
    stop=clock();
    duration=((double)(stop-begin))/CLK_TCK;
    printf("Tick for recursiveSequential:%lf,Time for recursivesequential:%lf\n",(double)(stop-begin),duration);
    return 0;
}
```

Listing 1: Source C Code

6 Declaration

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