Performance Measurement (Search)

2025-03-11

Chapter 1: Introduction

Searching is a fundamental operation in computer science, used to locate an element within a dataset. This project focuses on searching for a number N in a sorted list of integers from 0 to $N-1$, where N is guaranteed not to be in the list. This represents the worst-case scenario for many search algorithms.

The project involves implementing and analyzing two search methods: **sequential search** and **binary search**. Each method will be implemented in both **iterative** and **recursive** forms. By comparing their execution times across various input sizes (N), we aim to understand their efficiency and limitations in handling large datasets.

Chapter 2: Algorithm Specification

- ### specifications of main data structures

This project utilizes a one-dimensional array `a` to store a sorted list of integers, allowing efficient access and manipulation of elements through indexing. The array structure is particularly useful for implementing search algorithms, as it enables direct element retrieval. Additionally, the variables `left` and `right` are employed to define search boundaries, ensuring structured and efficient traversal, particularly in binary search.

- ### Sequential Search

Sequential search is a straightforward method that scans each element of the list one by one until the target is found or the list ends. Given that N is not in the list, this search will always traverse the entire dataset.

Iterative Implementation:

Algorithm: This algorithm scans the array from the `left` index to the `right` index, comparing each element in the search boundaries with the target value. If a match is found, it returns the `index`. Otherwise, it returns `-1`.

Pseudo-code:

```
Function Iterative_Version_Of_Sequential_Search(array,target,left,right):
    for each index in [left,right]:
        if array[index]==target:
            return index
    return -1
```

Recursive Implementation:

Algorithm: This algorithm searches for the target value by recursively checking first element within the search boundary. If a match is found, the index is returned. Otherwise, the function calls itself with updated search range `[left+1,right]`, continuing the search until `left` exceeds `right`, at which point `-1` is returned to indicate that the target is not in the array.

Pseudo-code:

```
Function Recursive_Version_Of_Sequential_Search(array,target,left,right):
    if left>right:
        return -1
    if array[left] == target:
        return left
    return Recursive_Version_Of_Sequential_Search(array,target,left+1,right)
```

- ## Binary Search

Binary search is designed for sorted arrays. It repeatedly divides the search range in half, comparing the middle element with the target value. Since the list is sorted, this method eliminates half of the remaining elements at each step.

Iterative Implementation:

Algorithm: This algorithm divides the search range `[left,right]` in half, comparing the middle element `a[mid]` with the target value (`mid=(left+right)/2`). If the middle element matches the target, its index is returned. Otherwise, the search continues in the left of the array `[left,mid-1]` if target is smaller, or right half of the array `[mid+1,right]` if target is greater. This process repeats until the search range becomes empty.

Pseudo-code:

```
Function Iterative_Version_Of_Binary_Search(array,target,left,right):
    while left <= right:
        mid=(left+right)
        if array[mid]==target:
            return mid
        elif array[mid]<target:
            left=mid+1
        else:
            right=mid-1
    return -1
```

Recursive Implementation:

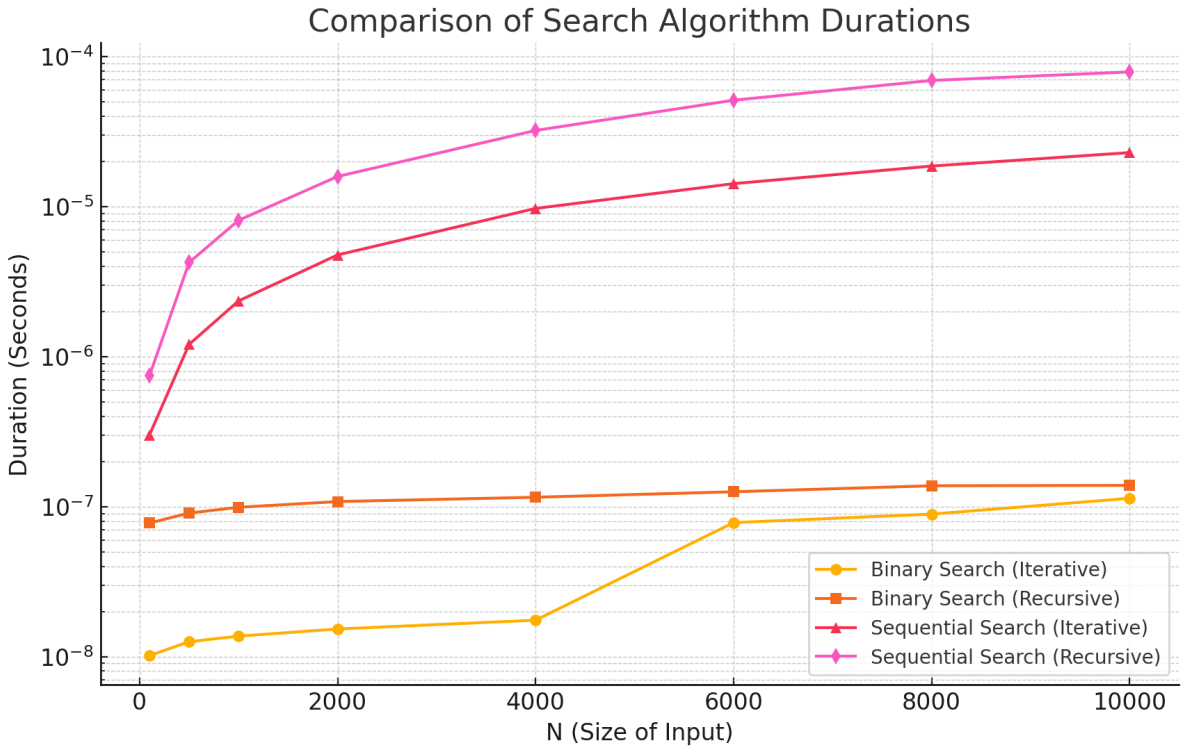
Algorithm: This algorithm divides the search range `[left,right]` in half, comparing the middle element `a[mid]` with the target value (`mid=(left+right)/2`). If the middle element matches the target, its index is returned. Otherwise, the function calls itself with an updated search range: if the target is less than the middle element, the search continues in the left half `[left,mid-1]`; if the target is greater, the search proceeds in the right half `[mid+1,right]`. This process repeats until the target is found or the search range becomes empty.

Pseudo-code:

```
Function Recursive_Version_Of_Sequential_Search(array,target,left,right):
    if left>right:
        return -1
    mid=(left+right)
    if array[mid]==target:
        return mid
    elif array[mid]<target:
        return Recursive_Version_Of_Sequential_Search(array,target,mid+1,right)
    else:
        return Recursive_Version_Of_Sequential_Search(array,target,left,mid-1)
```

Chapter 3: Testing Results

	N	100	500	1000	2000	4000	6000	8000	10000
Binary Search (iterative version)	Iterations(K)	100000000	100000000	100000000	100000000	100000000	100000000	50000000	50000000
	Ticks	1015	1256	1368	1527	1747	7811	4451	5670
	Total time(sec)	1.015	1.256	1.368	1.527	1.747	7.811	4.451	5.670
	Duration(sec)	1.015E-08	1.256E-08	1.368E-08	1.527E-08	1.747E-08	7.811E-08	8.902E-08	1.134E-07
Binary Search (recursive version)	Iterations(K)	50000000	50000000	50000000	50000000	50000000	50000000	50000000	50000000
	Ticks	3880	4522	4942	5397	5766	6272	6875	6920
	Total time(sec)	3.880	4.522	4.942	5.397	5.766	6.272	6.875	6.920
	Duration(sec)	7.760E-08	9.044E-08	9.884E-08	1.079E-07	1.153E-07	1.254E-07	1.375E-07	1.384E-07
Sequencial Search (iterative version)	Iterations(K)	10000000	5000000	1000000	1000000	1000000	500000	500000	100000
	Ticks	2997	6020	2347	4752	9704	7096	9295	2285
	Total time(sec)	2.997	6.020	2.347	4.752	9.704	7.096	9.295	2.285
	Duration(sec)	2.997E-07	1.204E-06	2.347E-06	4.752E-06	9.704E-06	1.419E-05	1.859E-05	2.285E-05
Sequencial Search (recursive version)	Iterations(K)	10000000	1000000	500000	500000	100000	100000	100000	100000
	Ticks	7447	4230	4044	7921	3211	5106	6916	7882
	Total time(sec)	7.447	4.230	4.044	7.921	3.211	5.106	6.916	7.882
	Duration(sec)	7.447E-07	4.230E-06	8.088E-06	1.584E-05	3.211E-05	5.106E-05	6.916E-05	7.882E-05



Observations:

- **Binary Search (Iterative)** is the fastest, maintaining a consistently low duration.
- **Binary Search (Recursive)** takes slightly longer but still remains efficient.
- **Sequential Search (Iterative)** is significantly slower than both binary search methods, especially as N increases.
- **Sequential Search (Recursive)** is the slowest and increases sharply as N grows.

Chapter 4: Analysis and Comments

Time Complexity

- **Sequential Search:**
  - **Iterative Implementation**:
    - Recurrence Relation: $T(n) = n$
    - Time Complexity: $T(n) = O(n)$
  - **Recursive Implementation**:
    - Recurrence Relation: $T(n) = T(n-1) + 1$
    - Time Complexity: $T(n) = O(n)$

Conclusion: Both iterative and recursive implementations of sequential search have the same time complexity because they both check each element in the worst case.

- **Binary Search**:
  - **Iterative Implementation**:
    - Recurrence Relation: $T(n) = T(n/2) + 1$
    - Time Complexity: $T(n) = O(\log n)$
  - **Recursive Implementation**:
    - Recurrence Relation: $T(n) = T(n/2) + 1$
    - Time Complexity: $T(n) = O(\log n)$

Conclusion: Both iterative and recursive implementations of binary search have the same time complexity because they both reduce the problem size by half in each step.

Space Complexity

- **Iterative Implementation of Sequential Search**: $O(1)$

  For only a constant amount of extra space is used for loop variables and indices.

- **Iterative Implementation of Binary Search**: $O(1)$

  For only a constant amount of extra space is used for variables like `low`, `high`, and `mid`.

- **Recursive Implementation of Sequential Search**: $O(n)$

  For each recursive call adds a new layer to the call stack, and in the worst case, there are n recursive calls.

- **Recursive Implementation of Binary Search**: $O(\log n)$

For each recursive call adds a new layer to the call stack, but the maximum depth of the recursion is $\log n$ because the problem size is halved in each step.

Comments

Sequential Search is straightforward and works on any list, whether sorted or unsorted. However, its time complexity of $O(n)$ makes it inefficient for large datasets. In contrust, **Binary Search** is significantly faster with a time complexity of $O(\log n)$, but it requires the input list to be sorted. This makes binary search more suitable for scenarios where the data is static or can be preprocessed, while sequential search is better for dynamic or unsorted data.

Iterative implementation is generally more space-efficient than recursive ones because they avoid the overhead of the recursion stack. For sequential search, the iterative approach uses $O(1)$ space, while the recursive approach uses $O(n)$ space. Similarly, for binary search, the iterative approach uses $O(1)$ space, while the recursive approach uses $O(\log n)$ space. Iterative methods are often preferred in practice due to their lower space complexity and better performance, especially in languages or environments where recursion depth is limited. However, **Recursive implementation** can be more intuitive and easier to implement, particularly for algorithms like binary search that naturally lend themselves to a divide-and-conquer approach.

Appendix: Source Code (in C)

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

// Iterative version of Binary Search
int Iterative_Version_Of_Binary_Search(int a[],int N,int left,int right){
    int mid;
    while(left<=right){ // Keep searching as long as the left pointer is less
than or equal to the right pointer
        mid=(left+right)/2;
        if (a[mid]==N) {
            return mid; // If the element is found at the middle index, return
the index
        }
        else if (a[mid]<N) {
            left=mid+1; // If the middle element is smaller, move the left
pointer
        }
        else {
            right=mid-1; // If the middle element is larger, move the right
pointer
        }
    }
    return -1;// Return -1 if the element is not found
}

// Recursive version of Binary Search
int Recursive_Version_Of_Binary_Search(int a[],int N,int left,int right){
    if (left>right){
        return -1; // Base case: if the left pointer is greater than the right,
the element is not found
    }
    int mid=(left+right)/2;
    if (a[mid]==N) return mid; // If the element is found, return the index
```

```
    if (a[mid]<N){
        return Recursive_Version_Of_Binary_Search(a,N,mid+1,right); // If the
middle element is smaller, search the right half recursively
    }
    else{
        return Recursive_Version_Of_Binary_Search(a,N,left,mid-1); // If the
middle element is larger, search the left half recursively
    }
}

// Iterative version of Sequential Search
int Iterative_Version_Of_Sequential_Search(int a[],int N,int left,int right){
    int i;
    for (i=left;i<=right;i++){ // Loop through the array from left to right
        if (a[i]==N) {
            return i; // If the element is found, return the index
        }
    }
    return -1; // Return -1 if the element is not found
}

// Recursive version of Sequential Search
int Recursive_Version_Of_Sequential_Search(int a[],int N,int left,int right){
    if (left>right) return -1; // Base case: if the left pointer is greater than
the right, the element is not found
    if (a[left]==N){
        return left; // If the element is found at the left pointer, return the
index
    }
    return Recursive_Version_Of_Sequential_Search(a,N,left+1,right); // Recursive
call by moving the left pointer to the right

}

// Function to allow the user to choose the search algorithm
void Function_choice(int (**Function)(int a[], int N, int left, int right),char
*Func_Name){
    int choice;

    // Display the options for the user to choose the search algorithm
    printf("Choose search algorithm:\n");
    printf("0. exit\n");
    printf("1. Iterative version of Binary Search\n");
    printf("2. Recursive version of Binary Search\n");
    printf("3. Iterative version of Sequential Search\n");
    printf("4. Recursive version of Sequential Search\n");
    printf("Enter your choice (1-4): ");
    scanf("%d", &choice);
    printf("\n");

    // Set the function pointer to the chosen search algorithm
    switch(choice){
        case 1:{
            *Function=Iterative_Version_Of_Binary_Search;
            strcpy(Func_Name,"Iterative version of Binary Search");
            break;
```

```
        }
        case 2:{
            *Function=Recursive_Version_Of_Binary_Search;
            strcpy(Func_Name,"Recursive version of Binary Search");
            break;
        }
        case 3:{
            *Function=Iterative_Version_Of_Sequential_Search;
            strcpy(Func_Name,"Iterative version of Sequential Search");
            break;
        }
        case 4:{
            *Function=Recursive_Version_Of_Sequential_Search;
            strcpy(Func_Name,"Recursive version of Sequential Search");
            break;
        }
        default :{
            exit(0); // Exit the program if the user input 0 or other invalid
input
        }
    }
}

int main(){
    clock_t start,stop;
    double duration;
    int N,K,i,a[10005],cnt;
    char Func_Name[100];
    int (*Function)(int a[], int N, int left, int right);

    // Initialize the array with values from 0 to 10004
    for (i=0;i<10005;i++){
        a[i]=i;
    }

    while (1){
        Function_choice(&Function,Func_Name); // Get user's choice of algorithm
        printf("Enter list size (N): ");
        scanf("%d",&N); // Read the list size
        printf("Enter number of iterations (K): ");
        scanf("%d",&K); // Read the number of iterations

        // Start measuring time
        start=clock();
        for(i=0;i<K;i++){
            Function(a,N,0,N-1); // Call the chosen search algorithm
        }
        stop=clock(); // Stop measuring time

        duration=((double)(stop-start))/CLK_TCK;// Calculate total duration

        if (Function(a,N,0,N-1)!=-1){
            printf("Algorithm Error!\n"); // Check for any errors
        }
        else {
            // Print the results
```

```
            printf("%s: N=%d, K=%d, Ticks=%d, Total_time=%lf,
Duration=%.12lf\n\n",Func_Name,N,K,stop-start,duration,duration/K);
        }
    }

    return 0;
}
```

Declaration

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