Practical 1: Searching Algorithms

Student: Dhruv Sachdeva, sachdevadhruv023@ce.du.ac.in

Practical 1.1: Linear Search

Problem 1. *Write a C program to implement linear search algorithm. Repeat the experiment for different values of n where n is the number of elements in the list to be searched and plot a graph of the time taken versus n.*

Input: A sequence of n numbers has < a 1 , a 2,..., a n > stored in array A[1:n] and a value x .
Output: An index i such that x equals A[1:n] or the special value NIL if x does not appear in A.

Algorithm

Require: An array $A[1 \dots n]$ and a value x
Ensure: Index i such that $A[i] = x$ or NIL if not found
1: **for** $i \leftarrow 1$ to n **do**
2: **if** $A[i] = x$ **then**
3: **return** i
4: **end if**
5: **end for**
6: **return** NIL

Code for Linear Search:

```c
/* Linear Search Program in C */
#include <stdio.h>
#include <stdlib.h>
# include<time.h>
void generateRandomArray(int arr[],int n)
{
    for(int i=0;i<n;i++)
    {
        arr[i]=rand()%100000;//generate random integers
    }
}
int linearsearch(int arr[],int n,int x)
{
   int result=-1;
    for(int i=0;i<n;i++)
    {
        if(arr[i]==x)
        {
            result=i;
```

```
            break;
        }

    }
    return result;
}
int main(){
    int n,x;
    printf("Enter the number of Elements: ");
    scanf("%d",&n);
    int* arr=(int*)malloc(n*sizeof(int));
    if(arr==NULL)
    {
        printf("Memory alloation failed!\n");
        return 1;
    }

    generateRandomArray(arr, n);

    // printf("Enter the element to search: ");
    // scanf("%d", &x);

    clock_t start, end;
    double time_used;

    start = clock();   // Start
    int result;

    for (int i = 0; i < 1000; i++) {
    result = linearsearch(arr, n, arr[rand() % n]);
}


    end = clock();     // End

    time_used = ((double)(end - start)) / CLOCKS_PER_SEC/1000.0;

    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");

    printf("Time taken for linear search: %f seconds\n", time_used);
    free(arr);
    return 0;
}
```

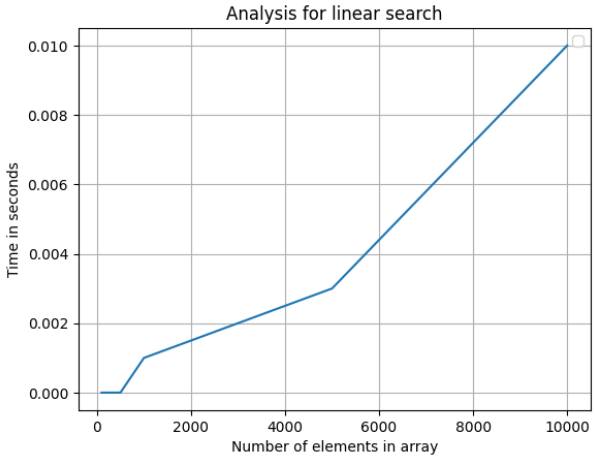

Figure 1: Graph of number of elements in array vs time

Algorithmic Analysis of Linear Search

The efficiency of the Linear Search algorithm can be analyzed as follows:

- **Best Case:** The best case occurs when the target element is found at the first position in the array.
$$T_{best}(n) = O(1)$$
- **Worst Case:** The worst case occurs when the target element is either at the last position or not present in the array at all.
$$T_{worst}(n) = O(n)$$
- **Average Case:** On average, the element may be found after scanning half of the list. Thus, the complexity remains linear:
$$T_{avg}(n) = O(n)$$
- **Space Complexity:** Linear Search only requires a few extra variables for iteration and comparison, hence:
$$S(n) = O(1)$$

Practical 1.2: Binary Search

Problem 2. *Write a C program to implement binary search algorithm. Repeat the experiment for different values of n where n is the number of elements in the list to be searched and plot a graph of the time taken versus n*

Input: A sequence of n sorted numbers has < a 1 , a 2,..., a n > stored in array A[1:n] and a value x .
Output: An index i such that x equals A[1:n] or the special value NIL if x does not appear in A.

Algorithm 1 Binary Search Algorithm

Require: Sorted array $arr[0 \ldots n-1]$, integer n, search element x
Ensure: Index of x in arr if present, otherwise -1

```
low ← 0
high ← n − 1
while low ≤ high do
    mid ← low + (high−low)/2
    if arr[mid] = x then
        return mid
    else if arr[mid] < x then
        low ← mid + 1
    else
        high ← mid − 1
    end if
end while
return −1
```

0.1 Code for Binary Search:

```c
    #include <stdlib.h>
#include <stdio.h>
#include<time.h>
// i will use inbuilt qsort so here is my definition of compare
int compare(const void* a, const void* b) {
    int x = *(int*)a;
    int y = *(int*)b;
    if (x < y) return -1;
    if (x > y) return 1;
    return 0;
}

void generateRandomArray(int arr[],int n)
{
        for(int i=0;i<n;i++)
        {
            arr[i]=rand()%10000;
        }
}
```

```c
int binary(int arr[],int n,int x)// n is len of arr and x is the required el
{
int low =0;
int high=n-1;
while(low<=high)
{
    int mid=low+(high-low)/2;
    if (arr[mid]==x) return mid;
    if (arr[mid]<x) low =mid+1;
    if(arr[mid]>x) high=mid-1;
}
return-1;
}

int main(){
    int n;
    printf("Enter the number of elements:\n");
    scanf("%d",&n);
    int * arr=(int*)malloc(n*sizeof(int));
    if(arr==NULL)
    {
        printf("Memory allocation failed\n");
        return 0;
    }
    srand(time(0));
    generateRandomArray(arr,n);
    qsort(arr, n, sizeof(int), compare);

    clock_t start,end;
    double time;
    int result;
    int reps=100000.0;
    start=clock();
    for (int i = 0; i < reps; i++)
    {
    result = binary(arr, n, arr[rand() % n]);
    }
    end=clock();
    time=((double)(end-start))/CLOCKS_PER_SEC/reps;
    if(result==-1) printf("Element not found!");
    else printf("Element is at index %d\n",result);
    printf("Time taken is %f nano-seconds\n",time*1e9);
    free(arr);
    return 0;
}
```

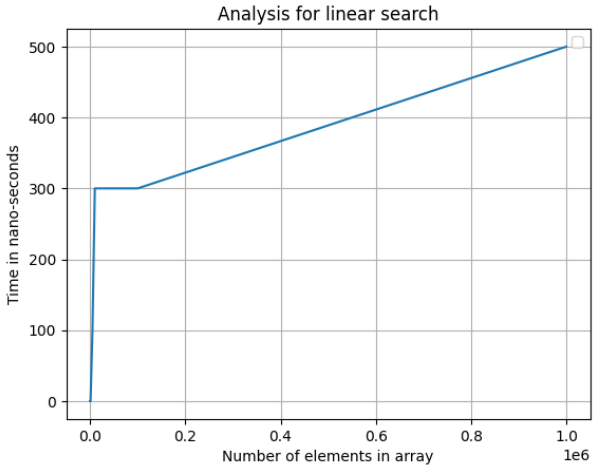

Figure 2: Graph of number of elements in array vs time(in nano-seconds)

Algorithmic Analysis of Binary Search

The efficiency of the Binary Search algorithm can be analyzed as follows:

- **Best Case:** The best case occurs when the target element is found at the middle index in the first comparison.

$$T_{best}(n) = O(1)$$

- **Worst Case:** In the worst case, the search space is repeatedly halved until only one element is left. This gives:

$$T_{worst}(n) = O(\log n)$$

- **Average Case:** On average, the element may be found after halving the search space multiple times, leading to:

$$T_{avg}(n) = O(\log n)$$

- **Space Complexity:** For the iterative implementation, the space requirement is constant, i.e.,

$$S(n) = O(1)$$

For the recursive implementation, additional space is required for the function call stack, leading to:

$$S(n) = O(\log n)$$

Conclusion

From the practical analysis and experiments, we conclude the following:

- **Linear Search:** The time taken by Linear Search increases linearly with the input size n. In this experiments, the execution time was recorded in **seconds**, since for large values of n (up to 10,000), the algorithm requires scanning through a significant portion of the array, resulting in noticeably higher runtimes.

- **Binary Search:** Binary Search demonstrates logarithmic time complexity. Even for $n = 100,000$, it performs fewer than 50 comparisons due to the repeated halving of the search space. The execution time was recorded in **nanoseconds**, as the computations are extremely fast and efficient compared to Linear Search.

Thus, while Linear Search is simple and effective for small datasets, Binary Search is vastly superior for large datasets, provided the input array is sorted.