Random point based Binary Search Algorithm

[1]Faizan Ahmad, [2] Mohammad Arif, [3]Faiyaz Ahmad

[1]Student, [2]Associate Professor, [3]Assistent Professor
[1]Dept. of Computer Science & Engg.
[1]Integral University, Lucknow, India

***Abstract* — In this paper represent a new concept of binary search algorithm using random check. A new formula is developed to find the Random Point in the list. And the most possible position of the element in the sorted list find by the algorithm. The complexity of algorithm depends on the gap among the elements. The algorithm can show better performance on large data set.**

***Index Terms* - Binary Search, Random Point.**

I. Introduction

Searching of element is an essential task in Computer Science. Only few searching algorithms are available in the literature. Widely used algorithms are Linear search and Binary search. Linear Search is the very simple algorithm in which we check the element one by one in the list. It is much easy to use but its time complexity is not good. In Binary Search the compulsion is that the elements should be in sorted order, and the searching process is based on finding the mid-point and the array is divided in two sub-arrays. The core of the Binary Search Algorithm is to find the mid-point, but if at first check the desired element is not found, the array is further divided into sub arrays and the same process is repeated again and again. When number of elements increases, complexity of Binary Search also increases. The suggested algorithm uses the weightage of the element at a given position to locate the element.

The method of algorithm is that, if there are n numbers in an array, placed in sorted order, then there is a possibility of a number to be at a particular location in that list.

II. Literature survey

The binary search plays a huge role in computer science. Different types of applications of binary search algorithm could be seen in this field. People have done many changes in the methodology according to their needs. In the paper by R.J. Lipton et al. [2], showed the goodness of Binary search. These new search methods can easily solve many typical problems of computer science. Eitan Zemel [3], has worked to find the global optimum. Random binary search is the work of author that results in optimum result. In the paper by Jan P. Allebach et al. [4], super order measure based model is combined for printer dot interactions. It is used within the iterative direct binary search (DBS) algorithm. An efficient method is shown by authors to evaluate the change in cost as the search increases. Results show the efficacy of the method. By Ahmed Tarek [1], Discrete Computational Structures can be easily examined by binary search algorithm. Logarithmic time complexity of the binary search makes it much efficient. It can be used to find the position of a key in a sorted list. Often, database applications are queried for two or more than two different key elements in one iteration. Binary search algorithm is changed to find more than one items in one iteration. In the paper, Q. L. Huang et al. [5], authors focused on collision problem in RFID. Using binary search of backtracking, an improved binary anti-collision search algorithm for radio frequency identification (RFID) system is demonstrated.

III. Proposed Work

Concepts used in the proposed algorithm are discussed below.

A. Methodology

In the paper, the new algorithm uses the list dividing method of old Binary Search algorithm. But the algorithms differ with each other as the new algorithm searches a Random Point instead of the middle point as in the old Binary Search. Another change have been made in the Binary Search Algorithm that the new algorithm checks both the right most and the left most element of the array.

The idea into the suggested algorithm is that if the numbers in given array are in order, then it is possible that an element is present at a particular position. That position can be found by calculating the AD into the array elements. AD acts as a key term to find the location of the element. The difference pattern creates three cases Best, Average and Worst case for the algorithm. Details discussed in the other section.

The Random Point (RP) is calculated using Average Difference (AD). Range (R) is calculated first to find the Average difference (AD). Difference between the right most and the left most element is called the Range (R).

$$\textit{Range (R) = Upper element - Lower element} \qquad (1)$$

The gaps are 1 less than total list elements (N).

$$\text{Number of Gaps} = N - 1 \qquad (2)$$

Now the R is divided by total gaps that results in AD.

$$AD = R / N\text{-}1 \qquad (3)$$

After evaluating the AD, the Random Point (RP) is calculated. For it, the difference between lower element and desired element is calculated. Then the difference is divided by the Average Difference. Finally the resultant is the Random Point (RP).

$$RP = (\text{Desired element} - \text{Lower element}) / AD \qquad (4)$$

The left and right most elements checking increase the probability of finding the desired element at first iteration.

B. *Proposed Algorithm*

The equation (1) is used to find the Random Point (RP) in the algorithm.
Following notations are used in the algorithm:

k :	The desired number.
a :	Name of the array.
low :	Index of lower-most element of the array.
up :	Index of upper-most element of the array.
tag :	flag.
rp :	index of Random Point
gap:	No. of Gaps among array elements.

The proposed algorithm is as follows:

```
Algorithm random_point_binary_search
while k ≥ a[low] and a[up] ≥ k and low ≤ up do
        if a[low] = k then
                print low+1
                tag ← 1
                break
        else if a[up] = k then
                print up+1
                tag ← 1
                break
        else
                rp ← (k-a[low])*gap/(a[up]-a[low]
                rp ← rp+low
                if a[rp] = k then
                    tag ← 1
                    print rp+1
                    break
                else if k < a[rp] then
                    up ← rp-1
                    low ← low+1
                else
                    low ← rp+1
                    up ← up-1
        end if
        gap ← up-low
 end while
 if tag ≠ 1 then
 print not found
```

end if

At the start of the algorithm, the while loop checks these conditions; first it checks if *k* is greater than or equal to left most element (*a [low]*) of the given array. Second, the right most element (*a[up]*) is greater than or equal to the *k*. The last condition is that the upper *(up)* & lower *(low)* indexes have jumped each other or not. The first two conditions are there to confirm if *k* is included in the array or not. The third condition looks that if *low* and *up* has jumped each other or not. If all the three conditions return true, the control will enter inside the loop. The second line of the algorithm checks if the left most number (*a [low]*) is *k* then it shows the position of the lowest element and terminates the loop. If the condition does not return true then it directly reaches to the 6th line and checks if the right most element (*a[up]*) and *k*, both are equal or not.

The control goes to the 11th line of the algorithm where the *Random Point (rp)* is calculated, If the *k* is not found at left most and the right most index location of the sorted list. In line 12, the *rp is set* according to the left most index (*low)* of the sorted list. In line 13, it is investigated that if the *a [rp]* and *k* both are equal or not. If not, line 17 looks whether the *k* is less than the *a[rp]*. If it returns true then the right most index is updated with the new value that is one less than the *rp*. The left index is also updated with the new value that is upward with one location.

If the line 17 does not return true, the left is updated to *rp+1* in line 21 & 22. *up is* also gets decreased. This update is done to arrange the lower and upper of the sub array. The loop terminates if any of the condition becomes false. In line 4, 8 and 14; the *tag* updates itself if the *k* is found. Line 26 looks the *tag*. If *tag* is not equal to 1 then the result is set to be not found.

IV. Experimental Result

A list of size 100 is taken to study the output pattern and the performance of the algorithm. We took 100 elements starting from 1 to 4900. If the numbers with equal difference is inserted to the array then the algorithms searches its each element at one iteration. At the position 50 if and abrupt increase is given then the algorithm shows bad results. If the gap among the elements are inserted according to the real time data then algorithm takes approximate 2.29 iterations to find an element, where binary search algorithm takes approximately 5.6 iterations to search the element.
Similar results could be seen on 1000 elements.

V. Performance Analysis

The suggested algorithm is based on Random Check, here the position of check point *rp* can be found anywhere within array list. This algorithm gets most possible position of the *k* on the basis of the *Average difference* between the list elements. Its been found if there is a sudden high difference across two numbers or, differences across list elements are random, it disturbs the difference and makes the worst case for algorithm.

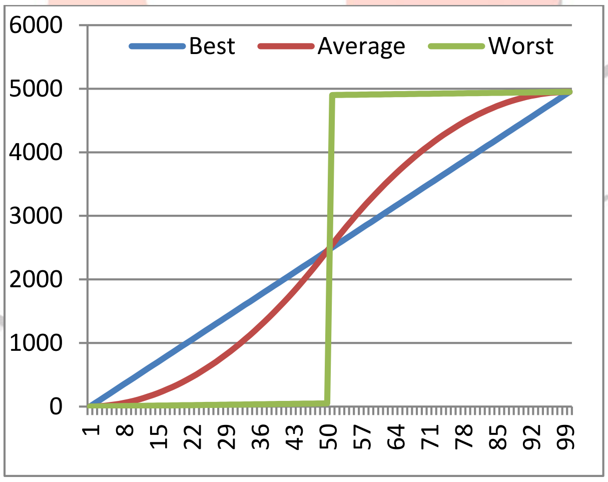

Fig. 1: Best, Average and Worst cases based on the differences between the list elements

The suggested algorithm is different from the traditional binary search algorithm, because the complications of traditional binary search depend on the probability that the k element is at the midpoint or anywhere else.

The suggested algorithm can perform better in the worst case of the traditional binary search if the differences across the two elements are equal or in a defined and limited range.

The performance of the suggested algorithm changes with the change in differences across the elements. Differences, that are almost close and distributed throughout the array makes the performance better.

The suggested algorithm performs best when the differences across the array elements are equidistant, but this situation is impractical. The miserable situation is when the elements increase with the difference of 1, which is a worst case for the algorithm.

VI. Comparative Analysis

When we compare the suggested algorithm with traditional binary search algorithm, the suggested algorithm works more efficiently than the traditional binary search in its Best and Average case which is noticeable. Graph for Best, Average and Worst cases are given below.

A. Best case analysis

Best case for the suggested algorithm does not depend on the position of the k element in the list, but on the differences between the elements. Suggested algorithm can search each element in single iteration if elements are equidistant.

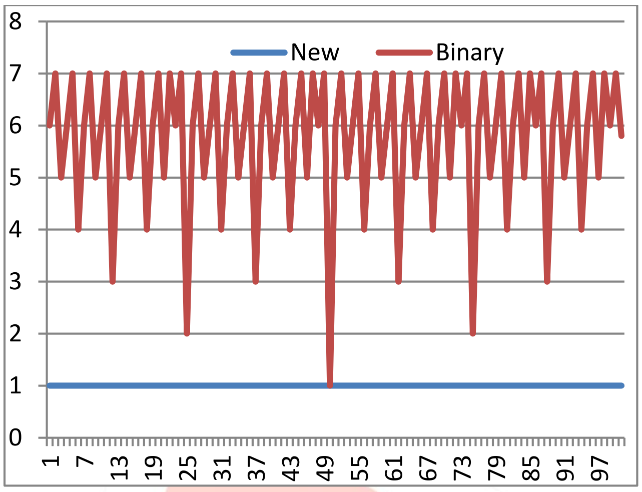

Figure 2: Best case of new suggested algorithm compare with behaviour of Binary search

Figure 2: X-axis depicts the number of elements in the array and Y-axis shows the number of iterations to search element k at a specific position. The behavior of binary search remains unchanged for fixed no. of elements, it will always first search for the midpoint and then move to left or right sub-list and so on, but the suggested algorithm acts according to difference across elements, if elements are equidistant then it will strike on the exact location of k element.

It is established that the average no. of iterations to search element k in an array using the suggested algorithm will be 1, but in case of binary search algorithm it will be more than 1 which increases logarithmically.

B. Average case analysis

For a predefined set of elements, Binary search algorithm works as usual, no matter where an element is placed and how much distant elements are.

While in the suggested algorithm differences across elements plays important role.

The average case of suggested algorithm occurs when the difference across the elements are unequal, but in a predefined range and distributed throughout the array.

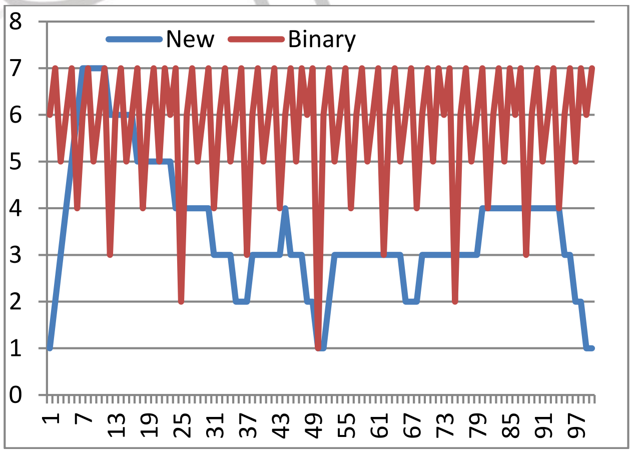

Figure 3: Average case of new proposed algorithm and the Binary

Figure 3, the graph shows the comparison between Binary search algorithm and the suggested algorithm. The results tested on 100 predefined elements shows that the suggested algorithm takes equal iteration to the binary search element at few locations, and it can also take more iteration in some cases but mostly it is taking less iteration than the Binary search algorithm at most of the locations. In comparison to the Binary search algorithm the suggested algorithm takes less iteration when we calculate average of all iterations.

On an average the suggested algorithm works better than the Binary search algorithm in its average case.

C. Worst case analysis

The suggested algorithm performs badly in its worst case. Worst case for the suggested algorithm is when the numbers increase in small distance from the starting but at the middle position of the array there is an abrupt extreme high increase in the difference, and after that sudden difference the elements again start increasing in small distance.

This condition becomes worst case for the suggested algorithm.

Figure 4 shows the graph for the performance of suggested and binary search algorithm in the worst case.

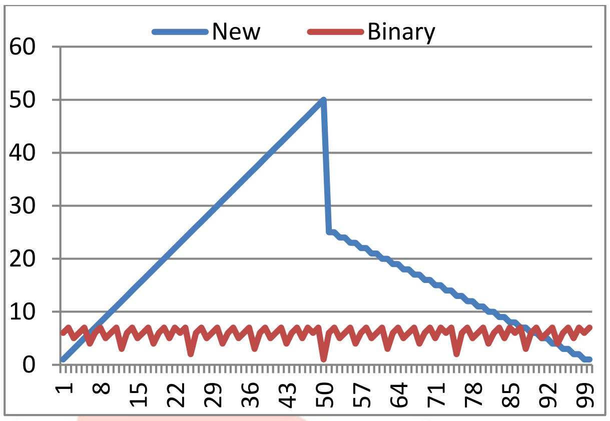

Fig. 4: No. of iteration in Worst case for the proposed algorithm

In this worst case condition algorithm begins replacing random point in a linear order upto n/2 in remaining portion of the array. After the center position the number of iterations diminishes to n/4 and starts decreasing by 1 after each two elements. This terrible nature makes the algorithm ineffective for the searching problems. Whereas the situation is impractical in which the algorithm works ineffectively.

VII. Conclusion and Future Scope

The suggested algorithm works better in case the differences across the array elements are not abruptly high. The suggested algorithm is useable where the elements are in almost equal or in a limited range. The real time data matches requirement of the suggested algorithm. In future the work on the suggested algorithm can be more equipped so as to match to the requirement of the technology.

References

[1] Ahmed Tarek-A New Approach for Multiple Element Binary Search in Database, INTERNATIONAL JOURNAL OF COMPUTERS,Issue 4, Volume 1, 2007

[2] R.J. Lipton and David Dobkin, On some generalization of binary serach, Sixth Annual ACM Symposium on theory of computing, Washington, April 1974.

[3] Eitan Zemel-Random Binary Search-A Randomizing algorithm for global optimization in R1*, Bonn Workshop on Combinatorial optimization, June 1984.

[4] Jan P. Allebach and Farhan A. Baqai-PRINTER MODELS AND THE DIRECT BINARY SEARCH ALGORITHM, 0-7803-4428-6198, 1998 IEEE.

[5] Q. L. Huang, X. L. Shi, F. Wei, L. Wang, and X. W. Shi-Novel Binary Search, Progress In Electromagnetics Research B, Vol. 9, 97–104, 2008.