Accepted Manuscript

New and improved search algorithms and precise analysis of their average-case complexity

Şahin Emrah Amrahov, Adnan Saher Mohammed, Fatih V. Çelebi

PII: S0167-739X(18)31930-7
DOI: https://doi.org/10.1016/j.future.2019.01.043
Reference: FUTURE 4736

To appear in: *Future Generation Computer Systems*

Received date : 11 August 2018
Revised date : 14 December 2018
Accepted date : 18 January 2019

Please cite this article as: Ş. Emrah Amrahov, A.S. Mohammed and F.V. Çelebi, New and improved search algorithms and precise analysis of their average-case complexity, *Future Generation Computer Systems* (2019), https://doi.org/10.1016/j.future.2019.01.043

New and improved search algorithms and precise analysis of their average-case complexity

Şahin Emrah Amrahov[a*], Adnan Saher Mohammed[b], Fatih V. Çelebi[c]
[a]Computer Engineering Department, Ankara University, 06830 Turkey
[b]Department of Computer Engineering Technologies, College of Engineering Technique,
Al-Kitab University, Kirkuk , Iraq
[c]Computer Engineering Dept. Faculty of Engineering and Natural Sciences, Ankara
Yıldırım Beyazıt University, Ankara, Turkey

Abstract

In this paper, we consider the searching problem over ordered sequences. It is well known that Binary Search (BS) algorithm solves this problem with very efficient complexity, namely with the complexity $\theta(\log_2 n)$. The developments of the BS algorithm, such as Ternary Search (TS) algorithm do not improve the efficiency. The rapid increase in the amount of data has made the search problem more important than in the past. And this made it important to reduce average number of comparisons in cases where the asymptotic improvement is not achieved. In this paper, we identify and analyze an implementation issue of BS. Depending on the location of the conditional operators, we classify two different implementations for BS which are widely used in the literature. We call these two implementations weak and correct implementations. We calculate precise number of comparisons in average case for both implementations. Moreover, we transform the TS algorithm into an improved ternary search (ITS) algorithm. We also propose a new Binary-Quaternary Search (BQS) algorithm by using a novel dividing strategy. We prove that an average number of comparisons for both presented algorithms ITS and BQS is less than for the case of correct implementation of the BS algorithm. We also provide the experimental results.

---

*Corresponding author: phone: +90 312 2033300 / 1771; e-mail: *emrah@eng.ankara.edu.tr*

Keywords: Binary search, Ternary search, Searching algorithm, Average-case complexity.

1. Introduction

Searching and sorting problems are classical problems of computer science. Due to excessive increase in the amount of data in recent years, these problems keep attracting the attention of researchers. In our previous work [29], we have made a short summary of the related works about sorting algorithms published recently [9, 13, 14, 17, 18, 32, 33, 39] . The study [1[illegible]] conducted after our publication proposes two novel sorting algorithms, called as Brownian Motus insertion sort and Clustered Binary Insertion Sort. Both algorithms are based on the concept of classical Insertion Sort. Marszałek [26] describes how to use the parallelization of the sorting processes for the modified method of sorting by merging for large datasets.

Besides of these studies Woźniak et al [40] modify Merge Sort algorithm for large scale data sets. Marszałek [25] proposes a new recursive version of fast sort algorithm for large data sets. Woźniak et al [41] examine quick sort algorithm in two versions for large data sets. Dymora et al [11] calculate the rate of existence of long-term correlations in processing dynamics of the quicksort algorithm basing on Hurst coefficient. Napoli et al [31] propose the idea of applying the simplified firefly algorithm to search for key-areas in 2D images. Woźniak and Marszałek [42] use classic firefly algorithm to search for special areas in test images. Das and Khilar [10] propose a Randomized Searching Algorithm and compare its performance with the Binary Search and Linear Search Algorithms. They show that the performance of the algorithm lies between Binary Search and Linear Search. Ambainis et al [1] study the classic binary search problem, with a delay between query and answer. They give upper and lower bounds of the matching depending on the number of queries for the constant delays. Finocchi and Italiano [12] investigate the design and analysis of the sorting and searching algorithms resilient to memory faults. Chadha et al [6] propose a modification to the binary search algorithm in which it checks the presence of the input element at each iteration. Rahim et al [34] provide the experimental comparison the linear, binary and interpolation search algorithms by testing to search data with different length with pseudo process approach. Kumar [22] proposes a new quadratic search algorithm based on binary search algorithm and he experimentally shows that this algorithm better than binary search algorithm.

Carmo et al [5] consider the problem of searching for a given element in a partially ordered set. Bonasera et al [4] propose an adaptive search algorithm over ordered sets. Proposed by Mohammed et al [30] hybrid search algorithm on ordered datasets is similar to the adaptive search algorithm. Bender et al [2] develop a library sort algorithm, which is developed based on insertion sort and binary search (BS) algorithm.

It is well known that BS algorithm is one of the widely used algorithms in computer applications due to obtaining a good performance for different data types and key distributions. It works on the principle of the divide-and-conquer approach [37]. This algorithm is used in solving several problems. For instance, Gao et al [15] propose a scheduling algorithm for ridesharing using binary search strategy. Hatamlou [19]presents a binary search algorithm for data clustering. BS is a simple and understandable algorithm, although it may contain some tricks in implementation. Donald Knuth emphasized: Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky [21]. Most of the implementation issues in the binary search were described in the literature. Pattis [35] notes five implementation errors. The study [36] involves a program to compute the semi-sum of two integers. In turn, this approach solves the problem of overflow that happens in binary search for very large arrays. Bentley discusses some errors in the implementation of the binary search in the section titled the challenge of binary search [3].

In this paper, we discuss two different implementations of the BS algorithm, which we call as weak and correct implementations. We calculate an average number of comparisons for both implementations precisely. We discuss the TS algorithm which is known as slower than BS, and then we present an improved ternary search (ITS) algorithm which is faster than the correct implementation of the BS algorithm. We prove this fact by calculating an average number of comparisons for ITS algorithm precisely. Moreover, we offer a new searching algorithm called as Binary-Quaternary Search (BQS) algorithm. We calculate an average number of comparisons for the BQS algorithm and we show that this algorithm is better than the correct implementation of BS algorithm. Theoretically, BQS slightly shows more average comparisons number compared with presented ITS algorithm.

The rest of the paper is organized as follows: In section 2 we discuss the weak and correct implementation of the BS algorithm. In this section we also calculate average number of the comparisons for weak and correct implementation of the BS algorithm. In section 3 we discuss the TS algorithm. In Section 4 we propose ITS algorithm and we calculate average number of comparisons for this algorithm. In section 5 we develop a new searching algorithm BQS and we find precisely average number of comparisons for BQS algorithm. In section 6 we compare the implementations of the ITS and BQS algorithms. In section 7 we demonstrate experimental results and comparison of these searching algorithms. Finally, we summarize our results in section 8.

2. Binary Search and Its Two Different Implementations

In this section we discuss the weak and correct implementation of the BS algorithm. We also calculate average number of comparisons for both implementations. We take the correct implementation from the book [37]. The weak implementation we meet in many works, for example, see [7, 24, 28]. Table 1 contains the correct and the weak implementation that is used in this study. Difference between these two implementations occurs when the first "*if*" statement is made to search for the desired key (contains equality test only), and the second "*if*" statement is used to decide whether half (right or left) will be selected for the next iteration. In result of this difference, we have a different number of comparisons in each iteration. In regard, while the binary search used as a search function in the Binary Search Tree (BST) data structure, we noticed the same issue in BST widely is observed. For example, see the BST implementation in [27, 38]. Meanwhile, the author of [20] presented the correct implementation of BST for recursive version and the weak implementation of the iterative version of BST. This drawback decreases the search speed in the binary search tree as well.

2.1. Binary Search Weak Implementation Analysis (Average case)

Figure 1 shows the comparisons tree of the weak implementation of binary search (Table 1). The main reason that makes the weak implementation slower than the correct implementation is the cost of selecting the next half that contains the required key, whereas the algorithm consumes three comparisons to select both halves (right or left half). In other words, in Figure 1, the branching to both children nodes consumes three comparisons.

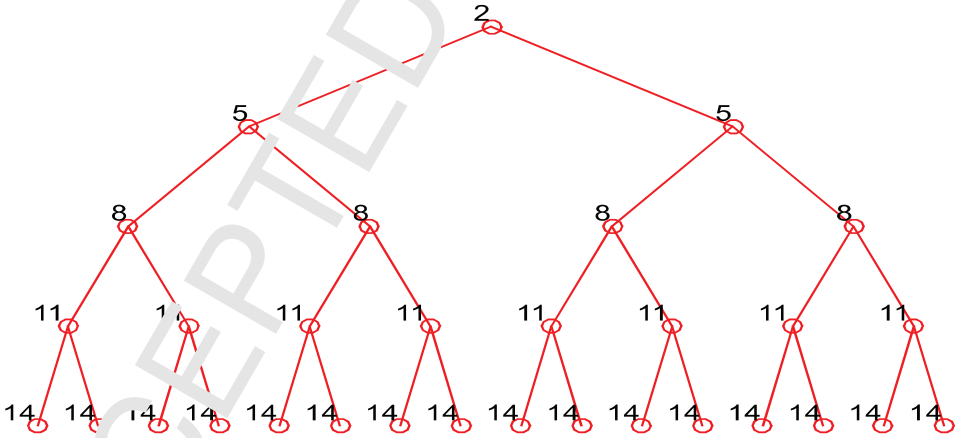


Figure 1: Comparison tree of binary search for the weak implementation.

Let $n = 2^k - 1$. Hence $k = \log_2(n+1)$. Let $C[j]$ be equal to a number of comparisons made for finding a jth element of the array. The average number

Table 1: Binary search correct and weak implementation comparison

Correct binary search implementation

```
template <typename T>inline int
    correctBS (T A[], int left,
    int right, T const& key)
{
  int mid ;
 while (left <= right)
  {
   mid = (left+ right) / 2;
   if (key < A[mid])
         right = mid - 1;
    else if (key > A[mid])
      left = mid + 1;
     else
return mid;
  } // end while
    return -1; // not found
}
```

Weak binary search implementation

```
template <typename T>inline int
   WeakBS (T A[], int left, int
   right, T const& key)
{
   int mid ;
 while (left <= right)
  {
    mid = (left+ right) / 2;
   if (key == A[mid])
          return mid;
    else if (key > A[mid])
                 left = mid + 1;
        else
    right = mid - 1;
   }
    return -1; // not found
}
```

of comparisons is $f(n) = \sum_{j=1}^{n} \frac{C[j]}{n}$. By the algorithm for one value (namely, for median) of j, we should make 2 comparisons (1 comparison for the base case “while left $\leq$ right”, 1 comparison for equality of key with median). For 2 values of j (for a median of left part and right part), we have to do 5 comparisons (1 comparison for the base case, 1 comparisons for equality key, 1 comparison for passing to left or right and plus previous comparisons.) For 4 values of j, similarly, we have to add 3 comparisons. Therefore, exactly for 2^{i-1} values of j we have to do $3i-1$ comparisons. Hence we have the following formula for the average number of comparisons:

$$f(n) = \sum_{i=1}^{k} \frac{(3i-1)2^{i-1}}{n} \tag{1}$$

Let

$$S_k = \sum_{i=1}^{k} i2^{i-1} \tag{2}$$

By multiplying by 2

$$2S_k = 2\sum_{i=1}^{k} i2^{i-1} = \sum_{i=1}^{k} i2^{i} \tag{3}$$

By subtracting (2) from (3) we obtain

$$S_k = -1 - \sum_{i=1}^{k-1} 2^i + k2^k = -2^k + 1 + k2^k \tag{4}$$

Hence,

$$S_k = (k-1)2^k + 1 \tag{5}$$

From the formula (5) for the average number of comparisons we have

$$f(n) = \frac{3}{n}\sum_{i=1}^{k} i2^{i-1} - \frac{1}{n}\sum_{i=1}^{k} 2^{i-1} = \frac{3}{n}S_k - \frac{1}{n}(2^k-1) = \frac{3}{n}[k2^k - 2^k + 1] - \frac{1}{n}2^k + \frac{1}{n} = \frac{3(n+1)}{n}\log_2(n+1) - 4 \tag{6}$$

Therefore,

$$f(n) = 3\log_2(n+1) + \frac{3\log_2(n+1)}{n} - 4 \tag{7}$$

2.2. Binary Search Correct Implementation Analysis (Average Case)

In this subsection, we calculate the average number of comparisons for correct binary search algorithm precisely. As in subsection 2.1, we suppose that $n = 2^k - 1$. We define also the functions $C[j]$ and $f(n)$ such as in subsection 2.1.

According to the algorithm for one value of j (for median) $C[j]$ is equal to 3. For one value of j (for a median of the left half) $C[j]$ is equal to 5. For one value of j(for a median of right half) $C[j]$is equal to 6. The values of $C[j]$ we

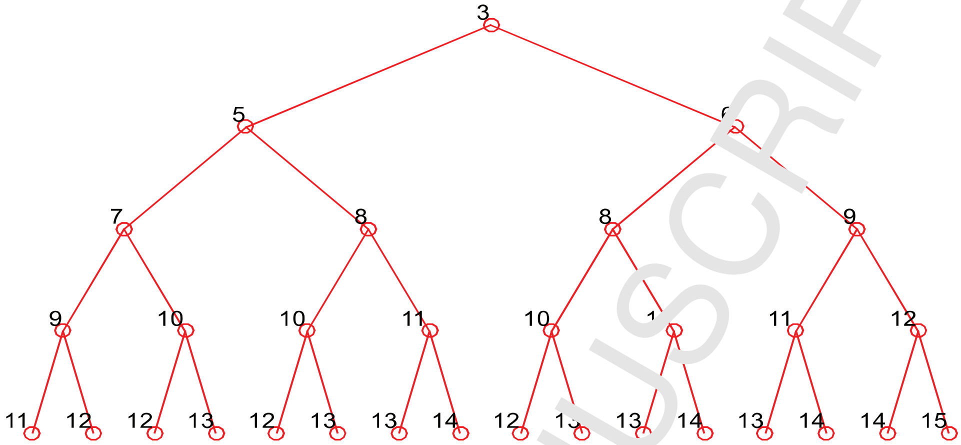


Figure 2: Comparison tree of binary search for the correct implementation.

show by the binary tree in Figure 2 for the value $n = 31$. We will call this tree by binary comparison tree (BCT).

According to correct BS algorithm initially, each iteration consumes one comparison by "*while*" statement. Then it may execute one or two comparisons in both "*if*" statements. If the first one is true, the algorithm goes to the left child node in the tree (Figure 2) and consumes two comparisons for the current iteration in total. However, if the second condition gets true, the algorithm goes to the right child node consuming three comparisons during the current iteration. Otherwise, the current node is equal to the required key, while this case also adds three comparisons to the total number of comparisons.

Briefly, as explained in Figure 2, walking to the left adds only two comparisons while walking to the right adds three comparisons. Moreover, we add three comparisons if we find the desired key in the current node. The number at each node represents the total number of comparisons when the algorithm terminated at this node.

We can observe that the values at i level change from $2i+3$ to $3i+3$ in the BCT.

Theorem 1. For any $0 \leq m \leq i$, number of values $2i+3+m$ at i level in the

BCT is equal to $\binom{i}{m}$.

Proof. We will prove by induction. For $i = 1$ it is true. Assume that it is true for all $k < i$. Let us calculate the number of $2i + 3 + m$ at i level for $0 \leq m \leq i$. For $m = 0$ we get the value $2i+3$ by adding 2 to the value $2(i-1)+3$ at $i-1$ level. By other words, we have only one value $2i+3$ at i level. Similarly, for the $m = i$ we get the value $3i + 3$ from the value $3(i-1)+3$ at $(i-1)$ by adding 3. If $0 < m < i$ we obtain the value $2i + 3 + m$ at i level from the value $2(i-1)+3+m$ at $i-1$ level by adding 2 or from the value $2(i-1)+3+m-1$ at $i-1$ level by adding 3.

By induction, the number of the values $2(i-1)+3+m$ at $i-1$ level is equal to $\binom{i-1}{m}$ and the number of the values $2(i-1)+3+m-1$ at $i-1$ level is equal to $\binom{i-1}{m-1}$. Therefore, by the property of binomial coefficients, the number of the values $2i+3+m$ at i level is equal to

$$\binom{i-1}{m-1} + \binom{i-1}{m} = \binom{i}{m}.$$

Now we can calculate an average number of comparisons for correct binary search algorithm. We have for the average number of comparisons $f(n)$ the following formula comparisons

$$f(n) = \sum_{j=1}^{n} \frac{C[j]}{n} = \frac{1}{n} \sum_{i=0}^{k-1} \sum_{m=0}^{i} \binom{i}{m} (2i+3+m)$$

Hence

$$f(n) = \frac{1}{n} \sum_{i=0}^{k-1} \left[(2i+3) \sum_{m=0}^{i} \binom{i}{m} + \sum_{m=0}^{i} m \binom{i}{m} \right]$$

Proposition 1.

$$\sum_{m=0}^{i} m \binom{i}{m} = i2^{i-1}$$

Proof. We have the formula $\begin{pmatrix} i \\ m \end{pmatrix} = \frac{i!}{m!(i-m)!}$

Therefore, for all $0 < m < i$,

$$\begin{pmatrix} i \\ m \end{pmatrix} m = \frac{i!m}{m!(i-m)!} = \frac{i!}{(m-1)!(i-m)!} = \frac{i(i-1)!}{(m-1)!((i-1)-(m-1))!} = i \begin{pmatrix} i-1 \\ m-1 \end{pmatrix}$$

For $m = 0$ and $m = i$ we have $0 \begin{pmatrix} i \\ 0 \end{pmatrix} = 0$ and $i \begin{pmatrix} i \\ i \end{pmatrix} = i$.

$$\sum_{m=0}^{i} m \begin{pmatrix} i \\ m \end{pmatrix} = 0 \begin{pmatrix} i \\ 0 \end{pmatrix} + \sum_{m=1}^{i-1} m \begin{pmatrix} i \\ m \end{pmatrix} + i \begin{pmatrix} i \\ i \end{pmatrix} = i \sum_{m=1}^{i-1} \begin{pmatrix} i-1 \\ m-1 \end{pmatrix} + i = i \sum_{m=0}^{i-1} \begin{pmatrix} i-1 \\ m \end{pmatrix} = i2^{i-1}.$$

Thus, we proved Proposition 1. Now we have

$$f(n) = \frac{1}{n} \sum_{i=0}^{k-1} \left[(2i+3)2^i + i2^{i-1}\right] = \frac{1}{n} \sum_{i=0}^{k-1} \left[5i2^{i-1} + 3.2^i\right] = \frac{5}{n} \sum_{i=0}^{k-1} i2^{i-1} + \frac{3}{n} \sum_{i=0}^{k-1} 2^i$$

By the formula (5) we have

$$S_{k-1} = (k-2)2^{k-1} + 1$$

Therefore we obtain

$$f(n) = \frac{5}{n}\left[(k-2)2^{k-1} + 1\right] + \frac{3}{n}(2^k - 1)$$

Since $2^k = n+1$, $k = \log_2(n+1)$ and $2^{k-1} = \frac{n+1}{2}$, so

$$f(n) = \frac{5}{n}\left[(\log_2(n+1) - 2)\frac{n+1}{2} + 1\right] + \frac{3}{n}.n$$

Finally, we have the formula

$$f(n) = \frac{5}{2}\log_2(n+1) + \frac{5\log_2(n+1)}{2n} - 2 \tag{8}$$

By comparing equation (7) and (8), we find that the average comparison number of weak implementation is greater than the number of correct binary search. Approximately, the average number of comparisons of weak implementation is equal to the worst-case comparison number of correct binary search. Consequently, binary search performance declined within this weak implementation.

Experimentally, the performance of weak implementation becomes slower when the cost of a single comparison operation increased. It happens for instance when the algorithm searches a list with long string keys. Let us discuss why the difference between correct and weak implementation occurs. If we look at the binary search again, we will find the issue occurs when the position of “*if*” statements have been altered. While nested “*if*” statements are widely used in most computer application, we will discuss the case of using the nested ”*if* ” statements and the influence of their occurrence probability on the performance of the whole program.

Let us examine the following two pseudo-code examples. Assume the loop repeats a nested “*if*” block for n times. We will examine how the position of “*if*” statement impacts the average number of comparisons. However, to get the best performance, the “*if*” statement with the highest probability of occurrence (the specified condition is true) must come first. Then it should be followed by the second highest probability “*if*” statement and so forth.

Example 1

```
for i=1 to n do
    if condition1 then                  ▷ 1 comparison
        statement 1                     ▷ Execution probability = 80%
    else if condition2 then             ▷ 2 comparisons till here
        statement 2                     ▷ Execution probability = 15%
    else                                ▷ 2 comparisons till here
        statement 3                     ▷ Execution probability = 5%
    end if
end for
```

Average Number of comparisons= $(1 * 0.8 + 2 * 0.15 + 2 * 0.05)n = 1.2n$.

Example 2

```
for i=1 to n do
    if condition3 then                  ▷ 1 comparison till here
        statement 3                     ▷ Execution probability = 5%
    else if condition2 then             ▷ 2 comparisons till here
        statement 2                     ▷ Execution probability = 15%
    else                                ▷ 2 comparisons till here
        statement 1                     ▷ Execution probability = 80%
    end if
end for
```

Average Number of comparisons= $(1 * 0.05 + 2 * 0.15 + 2 * 0.8)n = 1.95n$.
Example 1 represents the best performance which consumes $1.2n$ comparisons in average. Correspondingly, example 2 represents the weak performance which consumes $1.95n$ comparisons in average. The weak performance occurs as a result of the bad distribution of "*if*" statements.

3. The Ternary Search Algorithm

The ternary search is presented as an alternative to the binary search. This algorithm provides less number of iterations compared to binary search however it has a higher number of comparisons per a single iteration. In this section we explain this circumstance in detailed.

In literature, there are several studies presented for ternary search such as the analysis study in [23], the following pseudo-code (Algorithm 1) which is presented in [38] as a ternary search. In regard, there is a similar approach presented in [27].

Algorithm 1 The Ternary Search Algorithm

```
procedure TS( $array$, $left$, $right$, $X$)
    $array$ is the array that required to search
    $left$ is the index of left most element in $array$
    $right$ is the index of right most element in $array$
    $X$ is the element that we search for
    while $left < right$ do
        $Lci \leftarrow \lfloor \frac{2*left+right}{3} \rfloor$
        $Rci \leftarrow \lfloor \frac{left+2*right}{3} \rfloor$
        if $X = array[Lci]$ then
            return $Lci$
        end if
        if $X = array[Rci]$ then
            return $Rci$
        end if
        if $X \leq array[Lci]$ then
            $right \leftarrow Lci$
        else if $X \geq array[Rci]$ then
            $left \leftarrow Rci$
        else
            $left \leftarrow Lci + 1$
            $right \leftarrow Rci - 1$
        end if
    end while
    return -1                                    ▷ not found
end procedure
```

Total comparisons are 5 per iteration. Therefore, the maximum number of comparisons consumed by the ternary search is $5\log_3 n$, while it is $3\log_2 n$ in the binary search. Consequently, the comparison number in the ternary search is always higher than the comparison number in binary search algorithm because

$5\log_3 n > 3\log_2 n$.

4. Proposed Improved Ternary Search (ITS) Algorithm

The following pseudo-code (Algorithm 2) is the improved ternary search. This algorithm divides the length of the given array by three. Then it calculates the left cut index (*Lci)* and the right cut index(*Rci*). This method approximately divides the array into three equal parts. If the required key X is less than the key which is located at the Lci, the left third of the array will be contained X. Correspondingly, If X is greater than the key located at Rci, the right third of the array will be held X. Otherwise, the middle third holds the required key X. These operations repeat iteratively or recursively until the length of the scanned part of the array becomes less than or equal to 3. Then the algorithm uses a linear search to find X among remained keys to decide whether the search will finish successfully or unsuccessfully.

4.1. Improved Ternary Search Analysis(Average Case)

Improved ternary search decreases the average number of comparisons. This occurs because the algorithm continuously divides the array without searching for the required key until the length becomes less than or equal to 3.

Assume j is the position of the required element, $C[j]$ is the number of comparisons required to retrieve the element at j position. In each division process (iteration) there is only two possible states, if j at the left third, the algorithm consumes 2 comparisons to go to left part (1 comparison in “While” or base case, plus 1 comparison in the first “if” statement), so we have to add 2 to $C[j]$ in this case. If j residents at right or middle third, the algorithm requires 3 comparisons to go to the corresponding part (previous comparisons plus 1 for the second “if” statement), so we have to add 3 to $C[j]$ in this case.

The comparisons tree of the improved ternary search algorithm is shown in Figure 3. Walking to the left child node consumes two comparisons. Whereas walking to the middle or right child consumes three comparisons. However, the

Algorithm 2 Improved Ternary Search

```
procedure ITS( $array$, $left$, $right$, $X$)
    $array$ is the array that required to search
    $left$ is the index of left most element in $array$
    $right$ is the index of right most element in $array$
    $X$ is the element that we search for
    while $right - left > 2$ do
        $third \leftarrow \lfloor \frac{right-left}{3} \rfloor$
        $Lci \leftarrow left + third$
        $Rci \leftarrow right - third$
        if $X \leq array[Lci]$ then
            $right \leftarrow Lci$
        else if $X \geq array[Rci]$ then
            $left \leftarrow Rci$
        else
            $left \leftarrow Lci + 1$
            $right \leftarrow Rci - 1$
        end if
    end while                                   ▷ start linear search for remained items
    if $X = array[left]$ then
        return $left$
    else if $X = array[right]$ then
        return $right$
    else if $X = array[left + 1]$ then
        return $left + 1$
    else
        return $-1$                                                          ▷ not found
    end if
end procedure
```

improved ternary search uses a linear search (in last three "*if*" statements) to find X, if n or the remained number of elements is less than or equal to 3.

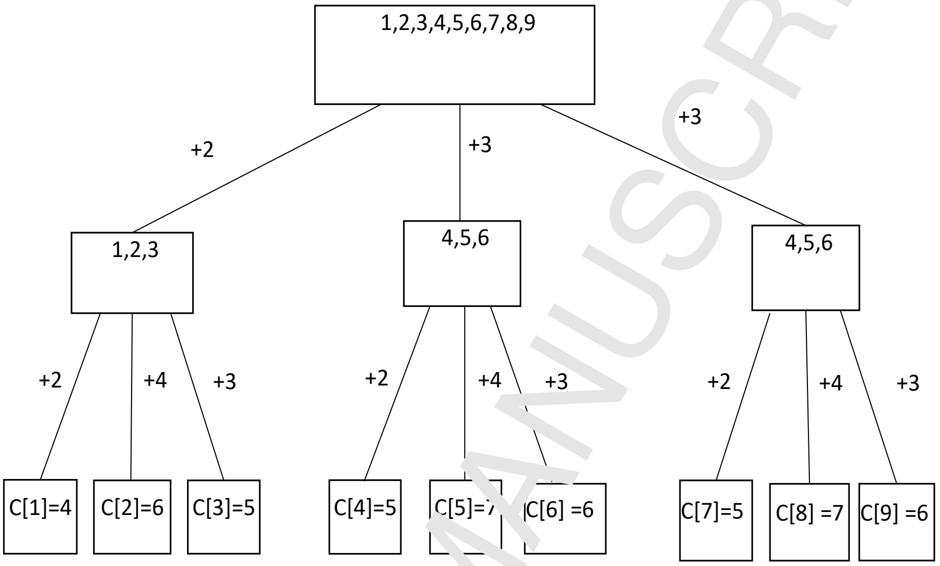


Figure 3: ITS Comparisons Tree

After the division process ends, the algorithm consumes 1 comparison to end the loop ("while" statement). Considering this comparison, linear search adds 2, 4 or 3 comparisons for the total number of comparisons that consumed in the division process before the algorithm terminated. Let $n = 3^k$. The minimum number of comparisons in the level $i\,(2 \leq i \leq k)$ of the comparison tree for the ITS algorithm is equal to $2i$. Therefore, in the best case the number of comparisons is equal to $2\log_3 n$. The maximum number of comparisons at level $i(2 \leq i \leq k)$ is equal to $3i + 1$. Hence, in the worst case the number of comparisons is equal to $3\log_3 n + 1$. To calculate the average case comparisons number, we have to calculate the total number of comparisons consumed by improved ternary search. From the comparison tree we can observe that we have the following recurrence for the ITS algorithm:

$$C[3^k] = 3C[3^{k-1}] + 8.3^{k-1}, \quad k \geq 2$$
$$C[3] = 9.$$

From here ,

$$
\begin{aligned}
C[3^k] &= 3C[3^{k-1}] + 8.3^{k-1} = 3(3C[3^{k-2}] + 8.3^{k-2}) + 8.3^{k-1} \\
&= 3^2C[3^{k-2}] + 2.8.3^{k-1} = 3^2(3C[3^{k-3}] + 8.3^{k-3}) + 2.8.3^{k-1} \\
&= 3^3C[3^{k-3}] + 3.8.3^{k-1} = ... \\
&= 3^{k-1}C[3] + 8(k-1)3^{k-1} \\
&= (8k+1)3^{k-1}
\end{aligned}
$$

Since $k = \log_3 n$ we obtain $C[n] = \frac{(8\log_3 n+1)n}{3}$. Therefore, average number of comparisons $f(n)$ is equal to $\frac{8}{3}\log_3 n + \frac{1}{3}$. Since $3^{15} > 2^{16}$, so $15\log_2 3 > 16$. From here we have the inequality $\frac{5}{2} > \frac{8}{3\log_2 3}$.

Average number of comparisons for correct BS and ITS are $f(n) = \frac{5}{2}\log_2(n+1) + \frac{5\log_2(n+1)}{2n} - 2$ and $g(n) = \frac{8}{3}\log_3 n + \frac{1}{3}$ correspondingly. Let us compare these functions.

$$f(n) > \frac{5}{2}\log_2(n+1) - 2 > \frac{5}{2}\log_2 n - 2$$

$$g(n) < \frac{8}{3}\log_3 n + 1 = \frac{8\log_2 n}{3\log_2 3} + 1$$

Thus, improved ternary search algorithm makes comparisons less than the correct implementation of binary search algorithm in average case for sufficiently large n.

Table 2 briefly compares the complexity of binary search and ternary search in term of comparisons number for the best, worst and average cases.

5. The Proposed Binary-Quaternary Search Algorithm

The proposed Binary-Quaternary search (BQS) is similar to ITS regarding the implementation. The main difference that BQS divides the length of the given array over four instead of three in ITS. Consequently, the behavior of the algorithm changed. Figure 4 shows the behavior of dividing technique in BQS.

When the required key X residents in the left quarter ($X \leq array[Lci]$), BQS sets (right=Lci) which excludes 75% of the length of the array for the next iteration. Likewise, when X residents in the right quarter, BQS sets (left=Rci).

Table 2: Complexity of Binary Search and Improved Ternary Search

Comparisons No.	**Correct Binary Search**	**Improved Ternary Search**
Best Case	*3*	$2\log_3 n = 1.82\ln(n)$
Worst Case	$3\log_2(n+1)+3 = 4.32\ln(n+1)+3$	$3\log_3 n + 1 = 2.73\ln(n)+1$
Average Case	$\frac{5}{2}\log_2(n+1) + \frac{5\log_2(n+1)}{2n} - 2 =$ $3.6\ln(n+1) + \frac{7.21\ln(n+1)}{2n} - 2$	$\frac{8}{3}\log_3 n + \frac{1}{3} = 2.42\ln(n) + 0.3$

In the case of X residents in the middle half (between Lci and Rci), BQS works like ordinary binary search by dividing the length over 2 . However, the main benefit of BQS is in each iteration, there is a chance of 50% to divide the given length over four consuming the same comparisons number in binary search. This approach is reducing the iteration number remarkably. In turn, it increases the performance of BQS.

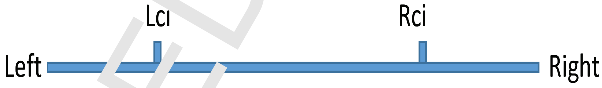


Figure 4: The Dividing technique of BQS

Algorithm 3 illustrates the pseudo-code of BQS. Initially, BQS calculates Lci which it indicates the end of the left quarter of the array and Rci denotes the beginning of the right quarter of the array.

Algorithm 3 Binary-Quaternary Search

```
procedure BQS( $array$, $left$, $right$, $X$)
    $array$ is the array that required to search
    $left$ is the index of left most element in $array$
    $right$ is the index of right most element in $array$
    $X$ is the element that we search for
    while $right - left > 3$ do
        $Quarter \leftarrow \lfloor \frac{right-left}{4} \rfloor$
        $Lci \leftarrow left + Quarter$
        $Rci \leftarrow right - Quarter$
        if $X \leq array[Lci]$ then
            $right \leftarrow Lci$
        else if $X \geq array[Rci]$ then
            $left \leftarrow Rci$
        else
            $left \leftarrow Lci + 1$
            $right \leftarrow Rci - 1$
        end if
    end while
    if $X = array[left]$ then                ▷ start linear search for remained items
        return $left$
    else if $X = array[right]$ then
        return $right$
    else if $X = array[left + 1]$ then
        return $left + 1$
    else if $X = array[right - 1]$ then
        return $right - 1$
    else
        return $-1$                          ▷ not found
    end if
end procedure
```

5.1. The Binary-Quaternary Search Algorithm Analysis (Average Case)

Let $n = 2^k$. Total number of comparisons for $n = 2, 4, 8, 16, 32$ is $5, 14, 46, 118, 298$ respectively. Let $C(n)$ be total number of comparisons. We observe that

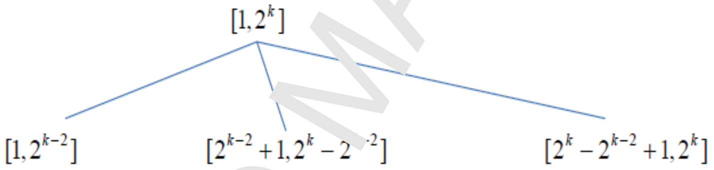
$$C(2^4) = 2C(2^2) + C(2^3) + 2^2.11 = 2.14 + 46 + 44 = 118$$
$$C(2^5) = 2C(2^3) + C(2^4) + 2^3.11 = 2.46 + 118 + 88 = 298$$

Now we will prove by induction that for all $k \geq 4$ we have the following recurrence.

$$C(2^k) = 2C(2^{k-2}) + C(2^{k-1}) + 2^{k-2}.11 \tag{9}$$

We have seen that the formula (9) is true for $k = 4$. Let the formula be true for all $4 \leq m < k$.

By BQS algorithm we have the following division for $n = 2^k$:



It means that we have the formula

$$C(2^k) = 2C(2^{k-2}) + C(2^{k-1}) + f(k)$$

By induction we have

$$C(2^{k-1}) = 2C(2^{k-3}) + C(2^{k-2}) + 2^{k-3}.11$$

$$C(2^{k-2}) = 2C(2^{k-4}) + C(2^{k-3}) + 2^{k-4}.11$$

Then

$$f(k) = 2.2^{k-4}.11 + 2^{k-3}.11 = 2^{k-2}.11$$

The last formula proves (9).

Now we will express $C(2^k)$ by $C(4)$ and $C(8)$. Let us define the following sequences:

$$\left.\begin{aligned} &x_{k+2} = x_{k+1} + 2x_k, \quad k \geq 4 \\ &x_4 = 2, \\ &x_5 = 2. \end{aligned}\right\} \tag{10}$$

$$\left.\begin{aligned} &y_{k+2} = y_{k+1} + 2y_k, \quad k \geq 4 \\ &y_4 = 1, \\ &y_5 = 3. \end{aligned}\right\} \tag{11}$$

$$\left.\begin{aligned} &z_{k+2} = z_{k+1} + 2z_k + 2^{k-2}, \quad k \geq 4 \\ &z_4 = 1, \\ &z_5 = 3. \end{aligned}\right\} \tag{12}$$

Theorem 2. For all $k \geq 2$ the formula

$$C(2^{k+2}) = x_{k+2}C(4) + y_{k+2}C(8) + 44z_{k+2} \tag{13}$$

holds.

Proof.

Since

$$C(2^4) = 2C(4) + C(8) + 44$$

So the formula is true for $k = 2$. Let the formula (13) be true for all $2 \leq m < k$. Then we have

$$C(2^{k+1}) = x_{k+1}C(4) + y_{k+1}C(8) + 44z_{k+1}$$

$$C(2^k) = x_kC(4) + y_kC(8) + 44z_k$$

By the formula (9)

$$\begin{aligned} C(2^{k+2}) &= 2C(2^k) + C(2^{k+1}) + 2^k.11 \\ &= 2x_kC(4) + 2y_kC(8) + 2.44z_k + x_{k+1}C(4) + y_{k+1}C(8) + 44z_{k+1} + 2^k.11 \\ &= (2x_k + x_{k+1})C(4) + (2y_k + y_{k+1})C(8) + 44(2z_k + z_{k+1} + 2^{k-2}) \\ &= x_{k+2}C(4) + y_{k+2}C(8) + 44z_{k+2} \end{aligned}$$

It means that the formula (13) is true.

Now we will solve recurrences (10),(11) and (12).Characteristic equation for all tree recurrences is the following quadratic equation.

$$r^2 - r - 2 = 0$$

From here we find $r_1 = 2$ and $r_2 = -1$. Therefore we have

$$x_k = c_1 2^k + c_2(-1)^k$$

and

$$y_k = d_1 2^k + d_2(-1)^k$$

From the initial value conditions for all $k \geq 4$ we find the formulas

$$x_k = \frac{2^k}{12} + \frac{2}{3}(-1)^k \tag{14}$$

and

$$y_k = \frac{2^k}{12} - \frac{1}{3}(-1)^k \tag{15}$$

Now we will seek a particular solution of the inhomogeneous equation (12) in the form

$$z_k = a(k-2)2^{k-2}$$

Then

$$z_{k+1} = a(k-1)2^{k-1}$$

$$z_{k+2} = ak2^k$$

If we substitute these values in the equation we find $a = \frac{1}{6}$. Then we find z_k in the form

$$z_k = e_1 2^k + e_2(-1)^k + \frac{(k-2)2^{k-2}}{6}$$

From the initial value conditions for all $k \geq 4$ we obtain finally

$$z_k = \frac{-2^k}{36} + \frac{(-1)^k}{9} + \frac{(k-2)2^{k-2}}{6} \tag{16}$$

From the formulas (14), (15), (16) for all $k \geq 4$ we find

$$\begin{aligned} C(2^k) &= x_k C(4) + y_k C(8) + 44 z_k \\ &= \left(\tfrac{2^k}{12} + \tfrac{2}{3}(-1)^k\right) C(4) \\ &+ \left(\tfrac{2^k}{12} - \tfrac{(-1)^k}{3}\right) C(8) \\ &+44 \left(\tfrac{-2^k}{36} + \frac{(-1)^k}{9} + \tfrac{(k-2)2^{k-2}}{6} \right) \end{aligned}$$

Since $C(4) = 14$, $C(8) = 46$ so we find total number of comparisons for the following formula:

$$C(2^k) = \frac{11k2^k}{6} + \frac{2^k}{9} - \frac{10}{9}(-1)^k$$

Therefore, the average number of comparisons is calculated by the formula:

$$\frac{C(2^k)}{2^k} = \frac{11k}{6} + \frac{1}{9} - \frac{10}{9}\frac{(-1)^k}{2^k}$$

Since $k = \log_2 n$ we see that the average number of comparisons for BQS algorithm is very close to the number $\frac{11 \log_2 n}{6}$.

By comparing BQS average comparisons number with the average case of correct BS and ITS (Table 2), we can see that BQS consumes fewer comparisons operations compared with BS, and slightly greater than ITS.

6. Implementation of ITS and BQS algorithms

The compiler used in the experimental work was configured to optimize the source code by default. However, most compilers optimize the division operation into a multiplication operation since the CPU consumes less time compared to the division operation. Furthermore, compilers optimize division or multiplication into shift operations when possible because the shift operation is much faster than the division and multiplication operations.

The C++ line in the ternary search ”Third = (right-left)/3;” and the line ”Quar = (right-left)/4;” in the BQS were compiled into assembly language by the compiler, as in Table 3 . In the ternary search, the compiler optimized the division over 3 by converting it into multiplication. Correspondingly, the compiler optimized the division over 4 into a shift operation in the BQS. Moreover, the assembly code in the BQS was smaller than that in the ITS. In another word, division over 4 is faster than division over 3. In turn, this increases the performance of the BQS compared to the ITS. In brief, the BQS showed better performance compared to the ITS when the cost of a comparison operation is less than the cost of division operation.

In this context, BS uses division over 2, so that compiler optimize this operation into a shift operation likewise BQS. While BQS has less average comparisons number compared to BS , so that BQS runs faster than BS for any type of data key.

7. Experimental Results and Comparisons

The experimental environment of this study is the same software and hardware configuration those were used in [30]. The experimental test has been done on empirical data that generated randomly using a C++ library [8]. Two types of generated data are used, a numeric array of 8-byte number (double) and text array of 100 characters' key length. The cost of a comparison process obviously effects on the performance of the algorithms under check. This cost is influenced by data type and hardware considerations. For instance, the computer needs more time to compare two strings of 100 bytes than two numbers of 8 bytes. Furthermore, the cost of any comparison increased when time to access the main memory increased. The other case that increases the cost of the comparison process is the external search. External search is the search when the array size is greater than the main physical memory or available memory. Certainly, access to secondary storage devices increases the time of the comparison process.

In previous sections, we calculated the average comparisons number for correct and weak BS, ITS and BQS algorithms precisely. To validate these results

Table 3: ITS and BQS implementation differences in assembly language

Ternary Search assembly code of C++ line: - Third= (right-left)/3;	BQ Search assembly code of C++ line: - Quar=(right-left)/4;
BaseAdd= starting address of Ternary_Search function	BaseAdd = starting address of BQ_Search function
BaseAdd+26: mov 0xc(%ebp),%eax	BaseAdd+26: mov 0xc(%ebp),%eax
BaseAdd+29: mov 0x10(%ebp),%edx	BaseAdd+29: mov 0x10(%ebp),%edx
BaseAdd+32: mov %edx,%ecx	BaseAdd+32: sub %eax,%edx
BaseAdd+34: sub %eax,%ecx	BaseAdd+34: mov %edx,%eax
BaseAdd+36: mov $0x55555556,%edx	BaseAdd+36: lea 0x3(%eax),%edx
BaseAdd+41: mov %ecx,%eax	BaseAdd+39: test %eax,%eax
BaseAdd+43: imul %edx	BaseAdd+41: cmovs %edx,%eax
BaseAdd+45: mov %ecx,%eax	BaseAdd+44: sar $0x2,%eax
BaseAdd+47: sar $0x1f,%eax	BaseAdd+47: mov %eax,%ebx
BaseAdd+50: sub %eax,%edx	
BaseAdd+52: mov %edx,%eax	
BaseAdd+54: mov %eax,%ebx	

experimentally, we measured the execution time of running each algorithm N times on N elements. In other words, we search for the all items of the tested list randomly then we record elapsed time. For figures (5,6 and 7), execution-time recorded in Y-axis, after each probe N increases by $N = N * 1.5$ until reaches the final array size (X-axis).

Experimental results show that the difference between weak and correct implementation is not detected in our test environment when the 8-byte key used. Figure 5 explains the experimental execution time for the weak and the correct implementation of binary search for 100-byte key length. The figure shows that there is a small gain in execution time for small size array and the gain increases when the array size increases.

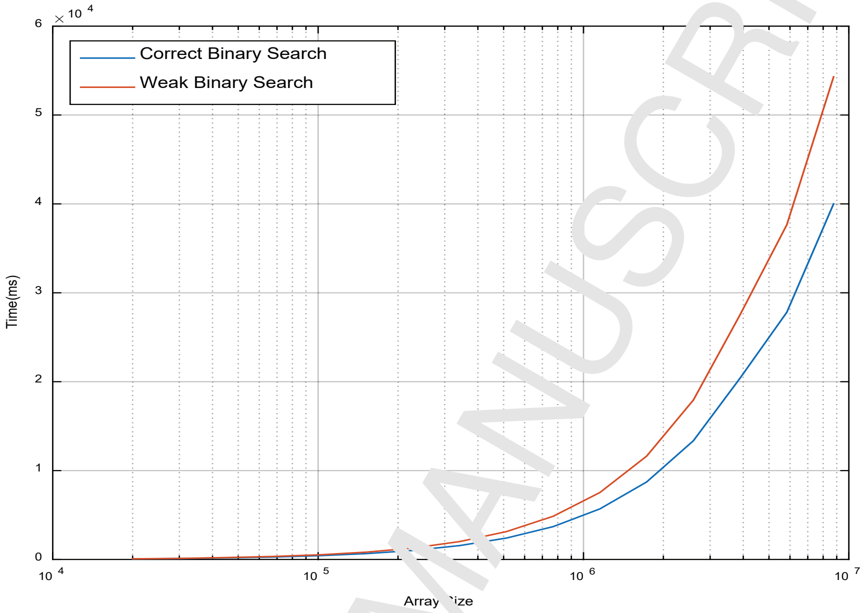


Figure 5: Binary search weak and correct implementation execution time.

Our presented algorithms (ITS and BQS) have less number of comparisons compared with the binary search. However, the drawback is they have more primitive operations compared to the binary search. One of the advantages is that the cost of the calculation of these variables does not depend on the data type or internal/external memory access operations. The other advantage is a limited number of variables involved in this computation, so the cache memory or CPU registers could hold these variables to reduce the access time to these variables, due to the frequent access to these variables.

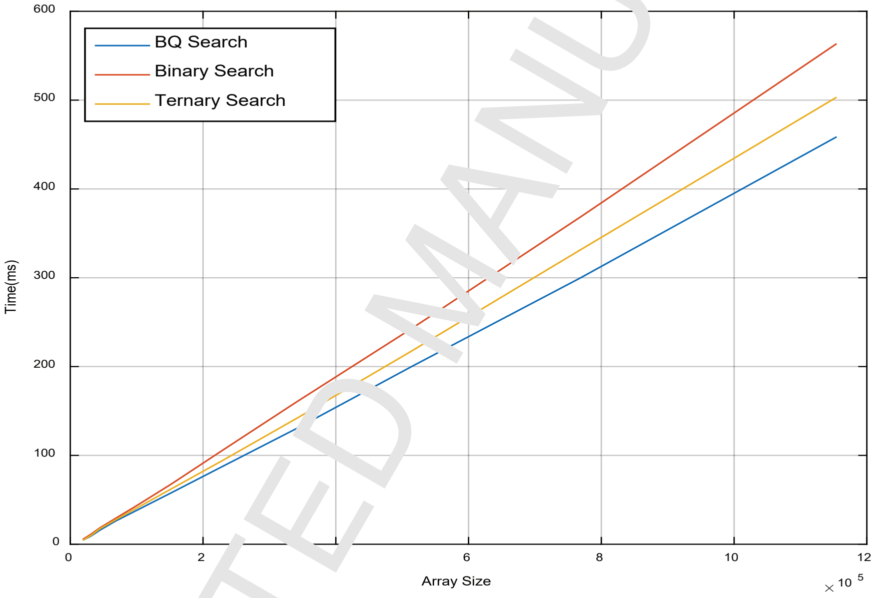


Figure 6: Execution time of BS, ITS and BQS for 8-byte key (double).

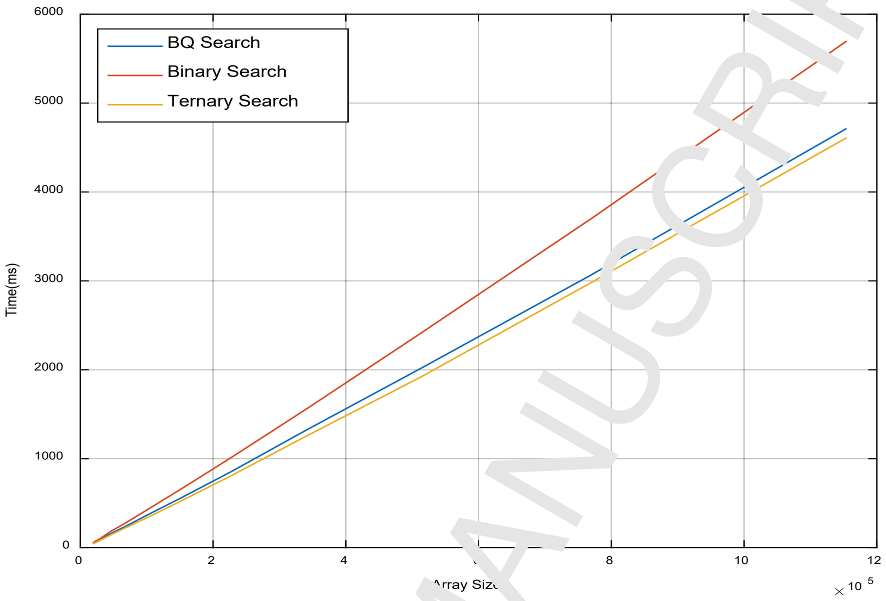


Figure 7: Execution time of BS, ITS and BQS for 100-byte key (string).

Figure 6 shows the experimental execution time of BS, ITS and BQS with a key of 8-byte double data type. We see the ITS execution time is less than the time consumed by the BS for moderate size array. The gain in the time increased when the size of the searched array increased. However, BQS offered better performance than ITS and correct BS in all array sizes.

Figure 7 shows the execution time for the same three algorithms runned with a key of 100-byte string data type. We see the ITS search execution time is less than the time consumed by the BS and BQS.

8. Conclusion

We examined the binary search algorithm in terms of comparisons. For BS we identified two implementations: weak and correct implementations. Our study explained that the correct implementation is faster than the weak implementation of BS.

We presented a new efficient improved ternary search algorithm (ITS). ITS has been analyzed and compared theoretically and experimentally with correct binary search. Comparison results showed that the improved algorithm is faster than the correct binary search. Our improvement on ternary search is obtained by reducing the number of comparisons per iteration.

Additionally, we proposed a new Binary-Quaternary search algorithm. The proposed algorithm is used to search ordered lists. The BQS is a divide-and-conquer algorithm and uses a new dividing technique, where it divides the given array length by 2 or 4 randomly. Theoretical analysis has shown that BQS has lower average comparison numbers than BS and slightly higher than ITS. On other hand, our experimental results showed that the BQS is faster than the ITS when the cost of a comparison operation is lower than the cost of a division operation.

Acknowledgements

We are grateful to the handling editor and anonymous reviewers for their careful reading of the paper and their valuable comments and suggestions.

References

[1] A. Ambainis, S. A. Bloch, D. L. Schweizer, Delayed binary search, or playing twenty questions with a procrastinator, *Algorithmica,* 32(4) (2002) 641-651.

[2] M. A. Bender, M. Farach-Colton, M. A. Mosteiro, Insertion sort is O(n log n), *Theory Comput. Syst.,* 397 (2006) 391–397.

[3] J. Bentley, *Programming pearls*, 2nd ed. Addison-Wesley Inc., 2000.

[4] B. Bonasera, E. Ferrara, G. Fiumara, F. Pagano, and A. Provetti, Adaptive search over sorted sets, *Journal of Discrete Algorithms,* 30 (2015) 128–133.

[5] R. Carmo, J. Donadelli, Y. Kohayakawa, and E. Laber, Searching in random partially ordered sets, *Theoretical Computer Science* 321(1) (2004) 41-57.

[6] A. R. Chadha, R. Misal, and T. Mokashi, Modified Binary Search Algorithm, *arXiv preprint arXiv:406.1677* (2014).

[7] S.-K. K. Chang, *Data Structures and Algorithms*, Series on. World Scientific, 2003.

[8] Cplusplus, "<random>- C++ Reference," 2011. [Online]. Available: http://www.cplusplus.com/reference/random/. [Accessed: 05-Nov-2016].

[9] A. Abu Dalhoum, T. Kobbay, A. Sleit, Enhancing QuickSort algorithm using a dynamic pivot selection technique, *Wulfenia* 19 (10) (2012) 543–552.

[10] P. Das, P. M. Khilar, A Randomized Searching Algorithm and its Performance analysis with Binary Search and Linear Search Algorithms. International Journal of Computer Science & Applications (TIJCSA), 1(11) (2013).

[11] P. Dymora, M. Mazurek, D. St[illegible]aka, Long-range dependencies in quick-sort algorithm, *Przeglad Elektrotechniczny* 90(1) (2014) 149-152

[12] I. Finocchi, G. F. Italiano, Sorting and searching in faulty memories, *Algorithmica,* 52(3) (2008), 309-332.

[13] M. L. Fredman, An intuitive and simple bounding argument for Quicksort, *Inform. Process. Lett.* 114 (3) (2014) 137– 139.

[14] M. Fuchs, A note on the quicksort asymptotics, *Random Structures Algorithms* 46 (4) (2015) 677–687.

[15] J. Ga[illegible], Y. Wang, H. Tang, Z. Yin, L. Ni, and Y. Shen, An Efficient Dynamic Ride-sharing Algorithm,*Computer and Information Technology (CIT), 2017 IEEE International Conference on.* IEEE, 2017.

[16] S. Goel, R. Kumar, Brownian Motus and Clustered Binary Insertion Sort methods: An efficient progress over traditional methods, *Future Generation Computer Systems* 86, (2018) 266-280.

[17] F. Grabowski, D. Strzalka, Dynamic behavior of simple insertion sort algorithm, *Fund. Inform.* 72 (1–3) (2006) 155–165.

[18] P. Hadjicostas, K. B. Lakshmanan, Recursive merge sort with erroneous comparisons, *Discrete Appl. Math.* 159 (14) (2011) 1398– 1417.

[19] A. Hatamlou, In search of optimal centroids on data clustering using a binary search algorithm, *Pattern Recognition Letters* 33(13) (2012): 1756-1760.

[20] N. Karumanchi, *Data Structures and algorithms made easy*, 5th ed. CareerMonk, 2017.

[21] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[22] P. Kumar, Quadratic Search: A New and Fast searching Algorithm (An extension of classical Binary search strategy, *International Journal of Computer Applications*, 65(14) (2013).

[23] A. Levitin, *Introduction to the design and analysis of algorithms*, 3rd ed. Pearson Inc., 2012.

[24] Lovro, Binary Search – topcoder.' [Online]. Available: https://www.topcoder.com/community/data-science/data-science-tutorials/binary-search/. [Accessed: 24-Jan-2017].

[25] Z. Marszałek, Novel Recursive Fast Sort Algorithm. In: Dregvaite G., Damasevicius R. (eds) Information and Software Technologies. ICIST 2016. Communications in Computer and Information Science, vol 639. Springer, Cham, 2016.

[26] Z. Marszałek, Parallelization of Modified Merge Sort Algorithm, *Symmetry*, *9*(9), (2017) 176.

[27] B. Miller and D. Ranum, *Problem Solving with Algorithms and Data Structures Using Python.* Franklin, Beedle & Associates, 2011.

[28] B. Miller and D. Ranum, *Problem Solving with Algorithms and Data Structures.* 2013.

[29] A. S. Mohammed, Ş. Emrah Amrahov, and F. V. Çelebi. Bidirectional Conditional Insertion Sort algorithm; An efficient progress on the classical insertion sort, *Future Generation Computer Systems* 71 (2017) 102-112.

[30] A. S. Mohammed, Ş. Emrah Amrahov, and F. V. Çelebi, Efficient hybrid search algorithm on ordered datasets, *arXiv preprint arXiv:1708.00964* (2017).

[31] C. Napoli, G. Pappalardo, E. Tramontana, Z. Marszałek, D. Poap, M. Woźniak, implified firefly algorithm for 2d image key-points search. In Computational Intelligence for Human-like Intelligence (CIHLI), 2014 IEEE Symposium on (pp. 1-8). IEEE, 2014.

[32] M. E. Nebel, S. Wild, Conrado Martínez, Analysis of pivot sampling in dual-pivot Quicksort: A holistic analysis of Yaroslavskiy's partitioning scheme, *Algorithmica* 75 (4) (2016) 632–683.

[33] R. Neininger, Refined Quicksort asymptotics, *Random Structures Algorithms* 46 (2) (2015) 346–361.

[34] R. Rahim, S. Nurarif, M. Ramadhan, S. Aisyah, W. Purba, Comparison Searching Process of Linear, Binary and Interpolation Algorithm, *Journal of Physics: Conference Series*, 930(1) (2017).

[35] R. E. Pattis, Textbook errors in binary searching, *ACM SIGCSE Bull.*, 20(1) (1988) 190–194.

[36] S. Ruggieri, On computing the semi-sum of two integers,' *Inf. Process. Lett.*, 87(2) (2003) 67–71.

[37] R. Sedgewick and P. Flajolet, *An introduction to the analysis of algorithms*, 2nd ed. Addison-Wesley, 2013.

[38] S. S. Skiena, *The Algorithm Design Manual*, 1(11), Springer London, 2008.

[39] S. Wild, M. E. Nebel, R. Neininger, Average case and distributional analysis of dual-pivot quicksort, *ACM Trans. Algorithms* 11(3) (201[illegible]) 22.

[40] M. Woź niak, Z. Marszałek, M. Gabryel, R. K. Nowicki, Modified Merge Sort Algorithm for Large Scale Data Sets. In: Rutkowski L., Korytkowski M., Scherer R., Tadeusiewicz R., Zadeh L.A., Zurada J.M. (eds) Artificial Intelligence and Soft Computing. ICAISC 2013. Lecture Notes in Computer Science, vol 7895. Springer, Berlin, Heidelberg, 2013.

[41] M. Woźniak, Z. Marszałek, M. Gabryel, R. K. Nowicki, Preprocessing Large Data Sets by the Use of Quick Sort Algorithm. In: Skulimowski A., Kacprzyk J. (eds) Knowledge, Information and Creativity Support Systems: Recent Trends, Advances and Solutions. Advances in Intelligent Systems and Computing, vol 364. Springer, Cham, 2016.

[42] M. Woźniak, Z. Marszałek Z, An Idea to Apply Firefly Algorithm in 2D Image Key-Points Search. In: Dregvaite G., Damasevicius R. (eds) Information and Software Technologies. ICIST 2014. Communications in Computer and Information Science, vol 465. Springer, Cham, 2014.

Adnan Saher Mohammed
College of Computer Science and Information Technology, Kirkuk University, Kirkuk 36013, Iraq
adnshr@gmail.com

Adnan Saher Mohammed received B.Sc. degree in 1999 in computer engineering technology from College of Technology, Mosul, Iraq. In 2012 he obtained M.Sc degree in communication and computer network engineering from UNITEN University, Kuala Lampur, Malasyia. He received his Ph.D degree in computer engineering from graduate school of natural sciences, Ankara Yıldırım Beyazıt University, Ankara, Turkey. His research interests include Computer Network and computer algorithms.

Fatih V. Çelebi
Ankara Yıldırım Beyazıt University, Faculty of Engineering and Natural Sciences, Computer Engineering Department, Ankara, Turkey

fvcelebi@ybu.edu.tr

Fatih Vehbi Çelebi obtained his B.Sc. degree in electrical and electronics engineering in 1988, M.Sc. degree in electrical and electronics engineering in 1996, and Ph.D. degree in electronics engineering in 2002 from Middle East Technical University, Gaziantep University and Erciyes University respectively. He is currently head of the Computer Engineering Department and vice president of Ankara Yıldırım Beyazıt University, Ankara-Turkey. His research interests include Semiconductor Lasers, Automatic Control, Algorithms and Artificial Intelligence.

Şahin Emrah Amrahov
Ankara University, Faculty of Engineering, Computer Engineering Department, Ankara, Turkey

emrah@eng.ankara.edu.tr

Şahin Emrah Amrahov received B.Sc. and Ph.D. degrees in applied mathematics in 1984 and 1989, respectively, from Lomonosov Moscow State University, Russia. He works as Associate Professor at Computer Engineering department, Ankara University, Ankara, Turkey. His research interests include the areas of mathematical modeling, algorithms, artificial intelligence, fuzzy sets and systems, optimal control, theory of stability and numerical methods in differential equations.

We propose improved ternary search (ITS) algorithm

We also propose a new Binary-Quaternary Search (BQS) algorithm

We discuss weak and correct implementations of the binary search (BS) algorithm

We calculate average number of comparisons for weak and correct implementations of the BS algorithm precisely

We calculate average number of comparisons for the ITS and BQS algorithms precisely