DEVELOPMENT OF SORTING AND SEARCHING ALGORITHMS

Ph. D Thesis By

ADNAN SAHER MOHAMMED

Department of Computer Engineering

December 2017

ANKARA
DEVELOPMENT OF SORTING AND SEARCHING ALGORITHMS

A Thesis Submitted to

The Graduate School of Natural and Applied Sciences of

Ankara Yıldırım Beyazıt University

In Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy
in Computer Engineering, Department of Computer Engineering

By

ADNAN SAHER MOHAMMED

December 2017

ANKARA
PH.D. THESIS EXAMINATION RESULT FORM

We have read the thesis entitled “Development of Sorting and Searching Algorithms” completed by Adnan Saher Mohammed under the supervision of Prof. Dr. Fatih V. ÇELEBİ and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Fatih V. ÇELEBİ
(Supervisor)

Prof. Dr. Şahin EMRAH
(Jury member)

Yrd. Doç. Dr. Hacer YALIM KELEŞ
(Jury member)

Yrd. Doç. Dr. Özkan KILIÇ
(Jury member)

Yrd. Doç. Dr. Shafqat Ur REHMAN
(Jury member)

Prof. Dr. Fatih V. ÇELEBİ
Director
Graduate School of Natural and Applied Sciences
ETHICAL DECLARATION

I hereby declare that, in this thesis which has been prepared in accordance with the Thesis Writing Manual of Graduate School of Natural and Applied Sciences,

- All data, information and documents are obtained in the framework of academic and ethical rules,
- All information, documents and assessments are presented in accordance with scientific ethics and morals,
- All the materials that have been utilized are fully cited and referenced,
- No change has been made on the utilized materials,
- All the works presented are original,

and in any contrary case of above statements, I accept to renounce all my legal rights.

Date: 20/12/2017  Signature:……………………………………

Name & Surname: Adnan ALAJEELI
ACKNOWLEDGEMENTS

I would like to gratefully acknowledge various people who have contributed in some way to accomplish this thesis.

First and foremost, there are no proper words to convey my deep gratitude and respect for my supervisor and the great mentor, Prof. Dr. Fatih Çelebi. The door to Prof. Çelebi’s office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but he steered me in the right direction whenever he thought I needed it.

My sincere thanks must also go to the members of my thesis advisory and exam committee: Prof. Dr. Şahin Emrah and Dr. Mustafa Yeniad. They generously gave their time to offer me valuable comments toward improving my work, especially Prof. Dr. Şahin Emrah, who has inspired me to become an independent researcher. Without him, this thesis would not have been completed or written.

I would like to express my very great appreciation for all people in the Graduate School of Natural Sciences at Ankara Yıldırım Beyazit Üniversitesi for their support, especially for my instructors throughout the academic courses which were a part of my Ph.D. dissertation, namely Prof. Dr. Abdullah Çavuşoğlu, Prof. Dr. Ünal Çamdali, Dr. Lami Kaya and Dr. Ali Tufail.

I owe my deepest gratitude to my uncle, Mr. Abdulla Mohammed and his family, for their wonderful services during my study years.

I want to acknowledge the study grant by the Salahuddin governorate department that allowed me to pursue my graduate school studies. I am indebted to my many colleagues who supported me in Salahuddin province: Mr. Ali Abdullah Najm, Mr. Akram Faysal, Mr. Bahaa Abbas, Mr. Humaydi Khalaf, Mr. Waleed Aziz, and Mr. Moayed Abdulwahab.

I would like to thank all my friends, those who expressed their good wishes and encouragement throughout my studies.

Last but not the least, I must express my profound appreciations to my family: My parents, my wife and my children for providing me endless support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them.

2017, 20 December    Adnan ALAJEELI
DEVELOPMENT OF SORTING AND SEARCHING ALGORITHMS

ABSTRACT

The increase in the rate of data is much higher than the growth in the speed of computers, which results in a heavy emphasis on sort and search algorithms in the research literature.

In this thesis, we propose a new efficient sorting algorithm based on the insertion sort concept. The proposed algorithm is called Bidirectional Conditional Insertion Sort (BCIS). It is an in-place sorting algorithm, and it has a remarkably efficient average case time complexity when compared with a standard insertion sort. By comparing our new algorithm with the QuickSort algorithm, BCIS shows faster average case times for relatively small arrays of up to 1500 elements. Furthermore, BCIS was observed to be faster than QuickSort within the high rate of duplicated elements even for large arrays.

We present a hybrid algorithm to search ordered datasets based on the idea of interpolation and binary search. The presented algorithm is called Hybrid Search (HS), which is designed to work efficiently on unknown distributed ordered datasets. Experimental results showed that our proposed algorithm has better performance when compared with other algorithms that use a similar approach.

Additionally, this study describes and analyses an unaddressed issue in the implementation of the binary search. In spite of this issue not affecting the correctness of the algorithm, it decreases its performance. However, the study presents a precise analytical approach to describe the behavior of the binary search in terms of comparisons number. With the help of this method, the complexity of the weak implementation is proved. Experimental results show that the weak implementation is slower than the correct implementation when a large sized search key is used. The presence of this implementation issue within other algorithms is also investigated.

Finally, we present two efficient search algorithms, the first of which is an improved implementation of the ternary search, and the second a new algorithm called Binary-Quaternary search (BQ search). BQ search uses a new efficient divide-and-conquer technique. Both proposed algorithms, theoretically and experimentally, show better performance when compared with binary searches. Although the proposed BQ search displays a slightly higher average comparisons number than the improved ternary search, experimentally the BQ search shows better performance compared with improved ternary searches under some conditions.

Keywords: bidirectional conditional insertion sort; binary search; ternary search, BQ search; hybrid search.
SIRALAMA VE ARAMA ALGORİTMALARININ GELİŞTİRİLMESİ

ÖZ

Bilgisayarlarla veri hızındaki artış bilgisayarların hızındaki büyümeden çok daha fazladır ve bu da araştırma literatüründeki sıralama ve arama algoritmalarının üzerinde yoğun bir şekilde durulmasına sebep olmaktadır.

Bu tezde, yerleştirmeli sıralama ‘insertion sort’ kavramına dayalı yeni verimli bir sıralama algoritması öneriyoruz. Önerilen algoritma Çift Yönlü Şartlı Yerleştirmeli Sıralama ‘Bidirectional Conditional Insertion Sort (BCIS)’ olarak adlandırılır. Bu algoritma bir yerinde sıralama algoritmasıdır ve standart yerleştirme sıralaması ile kıyaslandığında fevkalade verimli ortalama durum zaman karmaşıklığı sahiptir. Yeni algoritmanızı QuickSort algoritması ile kıyaslandığında, BCIS, 1500 ögeye kadar göreceli olarak küçük dizilerde daha hızlı ortalama durumları göstermektedir.

Enterpolasyon (ara değerlemesi) ve ikili arama fikrine dayanarak sıralanmış veri setlerini aramak için hibrit bir algoritma sunuyoruz. Sunulan algoritma Hibrit Arama (HA) olarak adlandırılır ve bilinmeyen dağılımlı sıralı veri setleri üzerinde verimli olarak çalışmak üzere tasarlanmıştır. Deneysel sonuçlar önerdiğimiz algoritmanın benzer bir yaklaşım kullanan diğer algoritmalarla kıyaslandığında daha iyı bir performansa sahip olduğunu göstermiştir.


Son olarak, iki adet verimli arama algoritması sunuyoruz. İlk iki aramanın gelişimi bir uygulamasıdır, ikincisi ise Binary-Quaternary search (BQ Arama) ‘İkili-Dörtlü Arama’ olarak adlandırılan yeni bir algoritmadır. BQ arama yeni verimli bir böl ve yönet teknği kullanmaktadır. Önerilen her iki algoritma da teorik ve deneysel olarak ikili aramalar ile kıyaslandığında daha iyı performans göstermektedir. Her ne kadar, önerilen BQ arama gelişmiş üçlü aramanın çok az daha yüksek ortalama karşılaştırma sayısını gösterse de, deneysel olarak BQ aramaları bazı koşullar altında gelişmiş üçlü aramalar ile kıyaslandığında daha iyı performans göstermektedir.

Anahtar Kelimeler: Çift Yönlü Şartlı Yerleştirmeli Sıralama; Ikili arama; üçlü arama, BQ arama; melez arama.
CONTENTS

PH.D. THESIS EXAMINATION RESULT FORM .............................................. ii
ETHICAL DECLARATION ........................................................................ iii
ACKNOWLEDGEMENTS .......................................................................... iv
ABSTRACT .............................................................................................. v
ÖZ ........................................................................................................... vi
NOMENCLATURE ................................................................................... ix
LIST OF TABLES .................................................................................... x
LIST OF FIGURES .................................................................................. xi

CHAPTER 1 - INTRODUCTION................................................................. 1
1.1 Asymptotic Computational Complexity ........................................ 2
1.2 Sorting Concepts ............................................................................ 3
  1.2.1 Definition of Sorting ............................................................. 3
  1.2.2 Sorting Algorithm Classification ......................................... 3
  1.2.3 Common Applications of Sorting ......................................... 6
1.3 Searching Concepts ........................................................................ 7
1.4 Thesis Contributions ..................................................................... 8
1.5 Thesis Organization ...................................................................... 9

CHAPTER 2 - BACKGROUND AND RELATED WORKS ..................... 10
2.1 Sorting Algorithms ......................................................................... 10
  2.1.1 Insertion Sort and Its Developments .................................. 10
  2.1.2 QuickSort Overview ......................................................... 14
2.2 Search Algorithms ........................................................................ 17
  2.2.1 Binary Search ..................................................................... 17
  2.2.2 Ternary Search and Quaternary Search .............................. 19
  2.2.3 Interpolation Search .......................................................... 19
  2.2.4 Hybrid (Interpolation-Binary) Search Approach ................. 20

CHAPTER 3 - BIDIRECTIONAL CONDITIONAL INSERTION SORT
(BCIS).................................................................................................. 22
3.1 The Proposed Algorithm BCIS .................................................... 22
3.2 BCIS Example .............................................................................. 26
3.3 Analysis of the Proposed Algorithm .......................................... 28
3.3.1 Average Case Analysis ................................................................. 29
3.3.2 Best Case Analysis ..................................................................... 35
3.3.3 Worst Case Analysis ................................................................. 37
3.4 Results and Comparison with Other Algorithms ....................... 39
  3.4.1 BCIS and classical insertion sort .............................................. 40
  3.4.2 BCIS and QuickSort ................................................................. 44

CHAPTER 4- DEVELOPMENT SEARCH ALGORITHMS FOR ORDERED
DATASET .......................................................................................... 47
  4.1 The Proposed Hybrid Search (HS) Algorithm ......................... 47
    4.1.1 Complexity of Hybrid Search .............................................. 49
    4.1.2 Hybrid Search Example ...................................................... 50
    4.1.3 Hybrid Search and Adaptive Search Comparisons .............. 52
    4.1.4 Results and experimental comparison ................................ 54
  4.2 Binary Search and Its Implementation Issue ......................... 60
    4.2.1 Binary Search Correct Implementation Analysis ............... 61
    4.2.2 Binary Search/Weak Implementation Analysis .................. 64
  4.3 Traditional Ternary Search ....................................................... 65
  4.4 Improved Ternary Search .......................................................... 67
    4.4.1 Improved Ternary Search Analysis .................................... 68
  4.5 The Proposed Binary-Quaternary Search ............................... 71
    4.5.1 BQ Search Analysis ............................................................. 73
  4.6 Implementation of Improved Ternary Search vs. BQ Search ...... 80
    4.6.1 Experimental results and comparisons ................................ 82
    4.6.2 Execution Time of Weak and Correct Binary Searches ........ 82
  4.7 Execution Time of Binary, Improved Ternary and BQ Searches .... 83

CHAPTER 5- CONCLUSION AND FUTURE WORK .............................. 85
  5.1 Conclusion .................................................................................. 85
  5.2 Future Work ................................................................................ 86

REFERENCES .................................................................................... 88

CURRICULUM VITAE ........................................................................ 97
## NOMENCLATURE

### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>Adaptive Search</td>
</tr>
<tr>
<td>BCIS</td>
<td>Bidirectional Conditional Insertion Sort</td>
</tr>
<tr>
<td>BQ search</td>
<td>Binary Quaternary Search</td>
</tr>
<tr>
<td>BS</td>
<td>Binary Search</td>
</tr>
<tr>
<td>BST</td>
<td>Binary Search Tree</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computer</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>EEBS</td>
<td>End-to-End Bi-directional Sorting</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>HS</td>
<td>Hybrid Search</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IntS</td>
<td>Interpolation Search</td>
</tr>
<tr>
<td>IS</td>
<td>Insertion Sort</td>
</tr>
<tr>
<td>IST</td>
<td>Interpolation Search Tree</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instructions Multiple Data</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>TS</td>
<td>Ternary Search</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large-Scale Integration</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 2.1 Insertion sort example ................................................................. 11
Table 4.1 Example of running HS on normally distributed array .................... 51
Table 4.2 Binary search and improved ternary search comparisons no. .......... 71
Table 4.3 Compilation of C++ instruction into assembly language ................. 81
LIST OF FIGURES

Figure 3.1 BCIS example .......................................................................................................................... 27
Figure 3.2 Best case example for less than 100 elements ................................................................. 35
Figure 3.3 Example of running BCIS on an already sorted array ..................................................... 36
Figure 3.4 Worst case example for arrays less than 100 elements ..................................................... 37
Figure 3.5 Example of running BCIS on a reverse sorted array ........................................................ 38
Figure 3.6 Number of comparisons and assignments for BCIS and IS ............................................. 40
Figure 3.7 BCIS/IS ratio for \( n < 6000 \) .............................................................................................. 41
Figure 3.8 BCIS/IS ratio for \( n > 6000 \) .............................................................................................. 42
Figure 3.9 BCIS/IS ratios with different data distributions ................................................................. 43
Figure 3.10 BCIS and QuickSort performance \( n < 2000 \) ................................................................. 44
Figure 3.11 Experimental BCIS/QuickSort ratio for \( n > 2000 \) ......................................................... 45
Figure 3.12 Experimental BCIS/QuickSort ratio for 50 duplicated elements ....................... 46
Figure 4.1 Dividing technique of the HS ............................................................................................... 49
Figure 4.2 Execution-time of IntS, BS, AS and HS for small array size with uniform distribution ................................................................. 55
Figure 4.3 Execution-time of IntS, BS, AS and HS for large array sizes with uniform distribution ................................................................. 56
Figure 4.4 Execution-time of BS, IntS, AS and HS for small array size with normal distribution ................................................................. 57
Figure 4.5 Execution-time of BS, IntS, AS and HS for large array size with normal distribution ................................................................. 57
Figure 4.6 Execution-time of BS, AS and HS for array small sizes with exponential distribution ................................................................. 59
Figure 4.7 Execution-time of BS, AS and HS for large array sizes with exponential distribution ................................................................. 59
Figure 4.8 Correct binary search comparisons tree .............................................................................. 62
Figure 4.9 Comparison tree of binary search for the weak implementation ......................................... 64
Figure 4.10 Improved ternary search comparisons tree ..................................................................... 69
Figure 4.11 BQ search dividing technique ............................................................................................. 72
Figure 4.12 BQ dividing tree .................................................................................................................. 74
Figure 4.13 Probability tree of BQ search ............................................................................................... 76
Figure 4.14 BQ average case iterations ................................................................................................. 78
Figure 4.15 Average comparisons numbers of the binary, improved ternary and BQ searches ................................................................. 80
Figure 4.16 Binary search weak/correct implementation execution time.............. 82
Figure 4.17 Execution time of binary, improved ternary and BQ searches for 8-byte double key................................................................. 84
CHAPTER 1

INTRODUCTION

Algorithms play an important role in the development process of computer science and mathematics. Sorting and searching are two of the essential problems in computer science. The importance of development sorting and search algorithms has increased as there is a massive amount of information accessed by an enormous number of users. The ability to process this information efficiently is usually based on the efficiency of sorting and searching. The high-level development of the computational infrastructure that we enjoy in the modern world arose through the aid of algorithms like these [1].

Sorting is a process in computer science which is commonly used for canonicalizing data. In addition to the primary function of sorting algorithms, many algorithms use different techniques to sort lists as a prerequisite step to reduce their execution time [2]. The idea behind using sorting algorithms by another algorithm is commonly known as the reduction process. A reduction is a method wherein one problem is transformed into another one that is easier than the first problem [1]. Consequently, the need for developing efficient sorting algorithms that invest in the remarkable development in computer architecture has increased.

Search algorithms are important and widely used in most computer systems. Searching for an item in an ordered list is an efficient operation in data processing. There are several search algorithms intended for ordered lists found in the literature. The binary search and interpolation search are the most traditional search algorithms used to search an already sorted dataset.

In this introductory chapter, we present an introduction to sorting and search fundamentals.
1.1 Asymptotic Computational Complexity

Asymptotic computational complexity is a mathematical expression that describes the run time of an algorithm [3]. There are three main asymptotic notations defining the complexity of algorithms [4,5]. For a given a function f(n):

- \(O\)-notation expresses the upper bound or asymptotic growth of the worst-case algorithm. The worst-case analysis anticipates the greatest amount of running time that an algorithm needs to solve a problem for any input of size \(n\). Formally, this is defined as:
  \[ O(f(n)) \text{ denoting the set of all } g(n) \text{ such that there exist positive constants } C \text{ and } n_0 \text{ with } |g(n)| \leq C f(n) \text{ for all } n \geq n_0. \]

- \(\Omega\)-notation describes the lower band or the best case of an algorithm. The best-case analysis anticipates the least amount of running time that an algorithm needs to solve a problem for any input of size \(n\). Formally, this is defined as:
  \[ \Omega(f(n)) \text{ denoting the set of all } g(n) \text{ such that there exist positive constants } C \text{ and } n_0 \text{ with } g(n) \geq C f(n) \text{ for all } n \geq n_0. \]

- \(\Theta\)-notation is used to state the tight bound that represents the average-case complexity of an algorithm. The average case analysis anticipates the average amount of running time that an algorithm needs to solve a problem for any input of size \(n\). The formal definition of these notations is as follows:
  \[ \Theta(f(n)) \text{ denoting the set of all } g(n) \text{ such that there exist positive constants } C_1, C_2 \text{ and } n_0 \text{ with } C_1 f(n) \leq g(n) \leq C_2 f(n) \text{ for all } n \geq n_0. \]

These computational complexity notations are not only used for time complexity analysis, they are also used to describe space complexity. Asymptotic notations are discussed in [3].
1.2 Sorting Concepts

In this section, we explore the essential features and the most important aspects of a sorting algorithm.

1.2.1 Definition of Sorting

Sorting is the procedure of repositioning a known set of objects in ascending or descending order according to specified key values belonging to these objects. Sorting is guaranteed to finish in a finite sequence of steps [6].

Formally, the output must satisfy two conditions:

- Each element is bigger than previous elements when non-decreasing sorting is intended.
- The output is a permutation, or reordering, of the input.

1.2.2 Sorting Algorithm Classification

Sorting algorithms used in computer science are often classified by:

- Time complexity, which is a way to describe the efficiency of algorithms concerning the number of elements asymptotically. The performance of algorithms is measured by the standard Big-Oh notation, which is used to describe the time complexity of an algorithm. The ideal behavior of a sort algorithm is $O(n)$ [7].
- Computational complexity may be expressed in terms of data swaps or moves for in-place algorithms.
- Space complexity determines the amount of memory that is required to run an algorithm. Algorithms are classified into two types depending on their space complexity: in-place or out-of-place. An in-place algorithm only needs a small amount of extra memory or a constant amount of space. The space complexity of such algorithms is denoted as $O(1)$. The drawback of an in-place algorithm is that the data becomes completely transformed, which in turn increases memory move or swap operations. In particular, swaps have an effect on performance for large data objects. In contrast, out-of-place algorithms usually
require extra memory depending on the size of the input dataset. These algorithms may avoid being changed on the positions of the original data set by working on a copy of the data set. This method reduces memory swap operations but it does increase memory usage.

- Stability refers to any sorting algorithm (being called stable) that preserves the relative order of equivalent elements as they appear in an unsorted input data set, whereas equivalent elements are elements with equal keys. For instance, we can assume that R1 and R2 are two records with equal keys, and R1 appears before R2 in the original data set. A sorting algorithm becomes stable if, and only if, R1 appears before R2 in the output dataset. Stability is not an issue when all keys of involved elements are different or the identical elements are indistinguishable, such as when the entire element is the key, including decimals or integer numbers.

- Internal or external sorting: The workspace of internal sorting algorithms is found entirely in the main physical memory (RAM). The physical memory of a computer system may be insufficient to hold the entire dataset that is required to be sorted. In such cases, an external sorting algorithm should be used to increase performance [8]. An external sort keeps the under-processing segment of the dataset in physical memory at any given time, whereas the entire dataset is stored in external storage devices. The basic external sorting algorithm uses the merging algorithm from a merge sort [9].

- Recursion: Some algorithms are either recursive or non-recursive, while others may contain features from both, such as a merge sort [10]. Obviously, recursive sorting algorithms use extra memory due to the nature of stack operations.

- Comparison or Non-comparison Sorting: A comparison sort algorithm inspects the dataset only by comparing or weighting two keys with a comparison operator. The lower band of any comparison based algorithm has been proved to be $\Omega(n \log_2 n)$ [3]. In the other words, any comparison based sorting algorithm must consume at least $n \log_2 n$ comparisons to sort an input dataset containing $n$ elements. Many non-comparison based sorting algorithms show better performance than the lower band $\Omega(n \log_2 n)$ of comparison based algorithms. Such algorithms
sort in linear time $O(n)$, but with some limitations on the length or type of key or dataset distribution. For instance, the radix sort and counting sort accept that the input dataset contains a small range of integers [11]. Moreover, the bucket sort works best when the input elements are distributed uniformly [12].

- **Hardware Architecture:** Sorting algorithms may be classified according to the hardware architecture for which they are designed. Most classical sorting algorithms are designed for serial processing systems with a single CPU and are known as sequential sorting algorithms. Furthermore, many parallel sorting algorithms are intended for parallel processing systems. Moreover, there are several types of parallel systems and many algorithms have been designed for each type [13], including the algorithm designed for MIMD machines in [14]. Similarly, the algorithm proposed in [15] is designed to exploit both SIMD instructions and thread level parallelism. Some algorithms are intended for GPU parallelism, such as the work presented in [16,17]. Furthermore, there are algorithms written to work on distributed systems [18]. On the other hand, there is a small number of sorting algorithms designed to be used with complex circuitry based on VLSI [19] or FPGA-based architecture [20]. Recently, Intel has introduced a technology that embeds an FPGA with a CPU. The new Xeon + FPGA accelerator platform enables new classes of algorithms for acceleration by implementing hybrid algorithms on a host and accelerator [21]. We believe this technology will open the door to a new generation of sorting algorithms.

- **Adaptivity:** A sorting algorithm is adaptive if it takes advantage of the existing order of the input data set. Usually, the time complexity of an adaptive sort algorithm with a partially sorted data set is considerably lower than its complexity when it is run on a random data set [22-24].

Among the many sorting algorithms, the choice of which is the best for an application depends on several factors such as size, data type and the distribution of the elements in a data set. Additionally, there are several dynamic influences on the performance of a sorting algorithm which can be briefed by the number of comparisons (for
comparison sorting), the number of swaps (for in-place sorting), memory usage and recursion [7,25].

### 1.2.3 Common Applications of Sorting

Sorting plays a major role in data processing and scientific computing. Sorting algorithms are widely used in a broad variety of applications, which include but are not limited to:

- **Searching**: A pre-sorted data set can search efficiently consuming only $O(\log n)$ time by using the binary search.

- **Multicriteria classification**: Operation research and artificial intelligence classify/sort the alternatives into predefined homogenous groups, which is an important issue for decision-makers. Classification algorithms are employed in numerous fields, such as medicine, pattern recognition, human resources management, production systems management and technical diagnosis, marketing, environmental and energy management and financial management and economics [26].

- **Priority queue**: This is an abstract data type that uses sorting to place the item of highest priority always at the front of the queue. If the first item is removed, the next highest priority item advances to the front. Priority queue use in various applications, such as Dijkstra’s shortest path algorithm, Prim’s algorithm, Data compression (used in Huffman codes) [1], pathfinding and graph traversals (A* Search Algorithm to approximate the shortest path in real-life situations). Additionally, it is used in operating systems for load balancing on the server and for interrupt handling.

- **More examples of applications** are included in [27], such as the closest pair, element uniqueness, frequency distribution (to find which items are duplicated the highest number of times in the data set), selection of kth largest/smallest item in a list and convex hulls. Other applications are mentioned in [1], such as commercial computing, numerical computations and string processing algorithms.
1.3 Searching Concepts

Information retrieval is one of the most common problems in computer science. On any computer, data are stored in data structures which are methods to organize the storage in both main memory and secondary memory [9]. There are a variety of data structures that have been designed to store data, including the array, the tree, the linked list, the hash table, the file, and so on. Data in these structures are frequently retrieved. The greatest concerns of data retrieval are the efficiency of data structures and computation speed, which is important for fast data retrieval [28]. For many programs, a search is the most time-consuming method [29]. Thus, there has been significant interest in the development of efficient search algorithms.

A search method is an algorithmic process of retrieving a specific object in a set of objects. The speed of search primarily depends on the storage structure of this set. However, even in the same storage structure, the use of an efficient searching algorithm also increases the speed of the entire search process.

Similarly to sort algorithms, search algorithms can be classified in several ways. First, search algorithms may be divided according to the memory type for which it has been designed. Completely identical with sort algorithms, there are internal search versus external search algorithms.

Second, search algorithms can be classified into static or dynamic searches. A static algorithm assumes that the contents of data structure are essentially unchanging. Therefore, the time consumed on setting up the data structure is disregarded. Conversely, dynamic search algorithms are designed to work on a data structure that are frequently subjected to insertion and deletion operations [29].

Search algorithms may be divided based on whether they use comparisons between keys or on the digital properties of the keys, not unlike the manner of classifying sorting algorithms into comparison-based and non-comparison-based algorithms. Finally, search algorithms can be classified into those that use the actual keys and others that work with transformed keys [29].
Each search algorithm is designed for certain data structures. For example, a linear search operates on a linear data structure, such as an array or linked list. The linear search is the simplest search algorithm which starts at the beginning and proceeds to the end, testing for a match at each element. This algorithm is efficient for very small data sets only. The advantage of the linear search is that the data structure is not required to be sorted.

Searches on ordered lists are much more efficient compared to linear searches. There are several algorithms intended for sorted lists, such as the binary search, the interpolation search, the ternary search, the Fibonacci search and the quaternary search. This thesis concentrates on searches over ordered lists only.

1.4 Thesis Contributions

This dissertation presents sort and search algorithms. The main contributions of this thesis are as follows:

- A new efficient sort algorithm called the Bidirectional Conditional Insertion Sort (BCIS), which is based on the principle of the classical insertion sort. BCIS is analyzed and compared with well-known algorithms. The results show that the average case complexity of BCIS is $O(n^{1.5})$. Therefore, BCIS is much faster than the classical insertion sort. On another hand, BCIS showed a performance faster than QuickSort for relatively small arrays.
- Development of a new hybrid search algorithm to search ordered datasets. The developed algorithm is called the Hybrid Search (HS). The presented algorithm is designed to search ordered lists within the unknown distribution. HS showed better performance when compared with similar solutions.
- Analyzing and addressing an implementation issue of the binary search. The analysis explained a weak implementation widely committed in the literature. This implementation issue made the weak implementation run slower than the correct implementation.
- Efficient implementation of the ternary search algorithm: The improved ternary search is analyzed and compared with the binary search. Theoretical
and experimental results show that the proposed ternary search is faster than the binary search.

- A new algorithm for searching ordered lists called the Binary-Quaternary search (BQ search). The theoretical and experimental results proved that the BQ search is faster than binary search. Moreover, BQ is faster than improved ternary search under some conditions.

1.5 Thesis Organization

Chapter One presents an introduction to sort and search concepts. Chapter Two presents a background and the related works. It discusses the related sort and search algorithms followed by listing the developments of these algorithms in the literature.

Chapter Three presents a new sort algorithm called the Bidirectional Conditional Insertion Sort (BCIS) followed by demonstrating an example on BCIS, analyzing the proposed algorithm theoretically, and presenting experimental results and comparisons with known algorithms.

Chapter Four presents four solutions for searches on ordered datasets. First, it proposes an algorithm called the Hybrid Search (HS), followed by comparing it with a similar solution. The experimental results of the hybrid search are presented afterwards. Second, this chapter highlights an implementation issue in the binary search and analyzes the impact of this issue on performance. Thereafter, it presents two new search algorithms called the improved ternary search and the Binary-Quaternary search. Then it shows the experimental results and comparisons of these algorithms.

Finally, Chapter Five presents the conclusion and future works.
CHAPTER 2

BACKGROUND AND RELATED WORKS

In this chapter, we examine the related sorting and search algorithms, and we explain the most significant developments on these algorithms in the literature.

2.1 Sorting Algorithms

Comparison based sorting algorithms may be classified into two groups according to their time complexity. The first group is $O(n^2)$, which contains the insertion sort, selection sort, bubble sort, etc. The second group is $O(n \log n)$, which is faster than the first group, and includes QuickSort, Mergesort and Heap sort [30]. As many sorting algorithms exist in the literature, we will explore two of the most popular sequential sorting algorithms, which are the Insertion sort and QuickSort.

Although parallel sorting algorithms are better than sequential algorithms in terms of performance, development of the basic sequential algorithms is still an important issue as they (sequential algorithms) are used implicitly inside parallel algorithms because parallel algorithms usually divide the given problem into smaller subproblems and solve each one sequentially in a single CPU simultaneously.

2.1.1 Insertion Sort and Its Developments

The insertion sort algorithm is considered to be one of the best algorithms in its family (the $O(n^2)$ group) due to its performance, stable algorithm, and simplicity and its being in-place [31]. Moreover, it is the fastest algorithm for small arrays up to 28-30 elements compared to the QuickSort algorithm. That is why it has been used in conjunction with QuickSort [32-35]. Algorithm 1 is the pseudo code of the insertion sort.
The example in Table 2.1 explains the behavior of the standard insertion sort. This algorithm consumes \( N - 1 \) passes and assumes that there is one sorted part. Initially, it consists of the leftmost element (for the ascending sort). Then, in each pass, it inserts an element from the unsorted part into the sorted part until every item is inserted. In order to insert an item into the sorted part, the insertion sort compares and shifts that element with adjacent elements until it reached the proper position.

**Table 2.1** Insertion sort example

<table>
<thead>
<tr>
<th>22</th>
<th>17</th>
<th>56</th>
<th>52</th>
<th>80</th>
<th>73</th>
<th>19</th>
<th>53</th>
<th>28</th>
<th>65</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>22</td>
<td>56</td>
<td>52</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>22</td>
<td>56</td>
<td>52</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>22</td>
<td>56</td>
<td>52</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>22</td>
<td>56</td>
<td>52</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>22</td>
<td>56</td>
<td>52</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>22</td>
<td>56</td>
<td>52</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>22</td>
<td>56</td>
<td>52</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>22</td>
<td>56</td>
<td>52</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>22</td>
<td>56</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>22</td>
<td>56</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>22</td>
<td>56</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>22</td>
<td>56</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>22</td>
<td>56</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>22</td>
<td>56</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>22</td>
<td>56</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>22</td>
<td>56</td>
<td>80</td>
<td>73</td>
<td>19</td>
<td>53</td>
<td>28</td>
<td>65</td>
</tr>
</tbody>
</table>

The example in Table 2.1 explains the behavior of the standard insertion sort. This algorithm consumes \( N - 1 \) passes and assumes that there is one sorted part. Initially, it consists of the leftmost element (for the ascending sort). Then, in each pass, it inserts an element from the unsorted part into the sorted part until every item is inserted. In order to insert an item into the sorted part, the insertion sort compares and shifts that element with adjacent elements until it reached the proper position.
The asymptotical analysis of insertion is well defined in the literature. The best case of an insertion sort is $O(n)$ when run on already sorted or almost sorted arrays; therefore, it is an adaptive sorting algorithm. The worst case of an insertion sort is $O(n^2)$, and the average case is also $O(n^2)$.

In this thesis, our proposed sorting algorithm is based on the standard insertion sort. Therefore, an understanding of the average case complexity for the insertion sort will give us a better understanding of our proposed algorithm. However, the average case complexity is calculated more precisely by [36] and [37] in terms of comparison and assignment (shift operations). The authors of the cited works presented the equations (Eqs. 2.1 and 2.2) which represent the average case analysis of the standard insertion sort for comparisons and assignments, respectively.

If we assume that we have a list of $n$ items, the insertion sort will require, in an average case, $n^2/4 + \Theta(n)$ comparisons and $n^2/4 + \Theta(n)$ assignments.

If we assume that a given element $X$ is initially in the $k$th position, it must shift to the right position $j$, where $j$ can be any value from 1 to $k$. The number of comparisons that are required to place $X$ at final position $j$ is $k - j + 1$. Therefore, on average, the number of comparisons to place the $k$th element is:

$$\frac{1}{k} \sum_{j=1}^{k} (k - j + 1) = \frac{1}{k} \left[ k^2 - \frac{k(k+1)}{2} + k \right] = \frac{k+1}{2}$$

For a dataset of $n$ elements, the average total cost of comparisons is:

$$T_{isc}(n) = \sum_{k=2}^{n} \left( \frac{k+1}{2} \right) = \frac{1}{2} \sum_{i=3}^{n+1} i = \frac{1}{2} \left[ \frac{(n+1)(n+2)}{2} \right] - \frac{1}{2} (1 + 2)$$

$$T_{isc}(n) = \frac{n^2}{4} + \frac{3n}{4} - 1 \quad (2.1)$$
The number of assignments required to shift the \( k \)th element to position \( j \) is \( k - j + 2 \) where \( j \) lies between 1 and \( k - 1 \). Similarly, the average total number of assignments is as follows:

\[
T_{\text{isa}}(n) = \sum_{k=2}^{n} \left( \frac{k + 3}{2} - \frac{k}{2} \right) < \sum_{k=1}^{n-1} \left( \frac{k + 4}{2} \right)
\]

\[
T_{\text{isa}}(n) = \frac{n^2}{4} + \frac{7n}{4} + 3 \quad (2.2)
\]

Several improvements on the major sorting algorithms have been presented in the literature [38-40].

Grabowski and Strzalka [41] investigated the dynamic behavior of the simple insertion sort algorithm and the impact of long-term dependencies in a data structure on sort efficiency. Biernacki and Jacques [42] proposed a generative model for rank data based on the insertion sort algorithm. The work that was presented by Bender et al. [43] is called the library sort or gapped insertion sort, which is a trade-off between the extra space used and the insertion time, so it is not an in-place sorting algorithm. Similarly, the insertion sort proposed by [44] was improved by reducing shift operations but with the aid of double-sized temporary array (O(2\(n\)) space complexity).

The enhanced insertion sort algorithm that is presented in [45] uses an approach similar to the binary insertion sort in [46]. While both algorithms reduced the number of comparisons and kept the number of assignments (shifting operations) equal to that in the standard insertion sort \( O(n^2) \), the binary insertion sort was preferred over the standard insertion sort when the cost of a comparison operation is significant [46].

Bidirectional insertion sort approaches are presented in [6,47]. The authors of these works attempted to produce a semi-sorted list in the preprocessing step by swapping the elements at analogous positions (position 1 with \( n \), position 2 with \( (n - 1) \) and so
on). Then they applied the standard insertion sort to the whole list. The main goal of this work was only to improve the worst-case performance of the insertion sort [30]. The doubly inserted sort presented in [48] needs on average \((n^2 - 6n + 8)/8\) passes to sort a list of \(n\) elements. Even though this is less than the standard insertion sort, it is still asymptotically \(O(n^2)\). The most significant advantage of the doubly inserted sort is that it only requires \(n/2\) steps in a reverse sorted array (worst case). On the other hand, the authors in [31] presented a bidirectional insertion sort, firstly exchanging elements using the same approach as [6,47], followed by starting from the middle of the array and inserting elements from the left and the right sides to the sorted portion of the main array. This method improves the performance of the algorithm such that it is efficient for small arrays typically sizes of 10-50 elements [31]. Similarly, a sorting algorithm called the End-to-End Bi-directional Sorting (EEBS) algorithm was proposed in [30]. In the first step, it swaps the front half of a list with the rear half depending on the values of the analogue elements. In the second phase, it compares two adjacent elements from the front and rear ends of the array. Then, EEBS swaps the items if required according to the order. The author of this work compared his algorithm with the standard bubble sort and selection sort. Although this algorithm displayed better performance, it still ran within time complexity equal to \(O(n^2)\).

Finally, the main idea of the work presented in [49] was based on inserting the first two elements of the unordered part into the ordered part during each iteration. This idea became slightly time efficient but the complexity of the algorithm was still \(O(n^2)\) [49].

2.1.2 QuickSort Overview

QuickSort is one of the most widely used general purpose sorting algorithms. It was presented by Hoare [50] in the early 1960s. QuickSort is an in-place, comparison-based, divide-and-conquer method and a recursive algorithm. The standard version of QuickSort is an unstable algorithm. Despite the slow worst-case running time of QuickSort being \(O(n^2)\), it is often the best practical choice for sorting as it is remarkably efficient on average [3]. QuickSort is classified as the most efficient algorithm in the sorting group of \(O(n \log n)\) time complexity.
The pseudo code of QuickSort is presented in Algorithm 2.

**Algorithm 2: Quick Sort**

1: procedure QUICKSORT( array, left, right)
2:     array is the array that required to sort
3:     left is the index of left most element in array
4:     right is the index of right most element in array
5: if length(array) > 1 then
6:      pivot ← select any element of array
7:     while left ≤ right do
8:          while array[left] < pivot do
9:              left ← left + 1
10:         end while
11:     while array[right] > pivot do
12:          right ← right − 1
13:    end while
14: if left ≤ right then
15:     swap array[left] with array[right]
16:     left ← left + 1
17:     right ← right − 1
18: end if
19: end while
20: QUICKSORT(array, first index, right)
21: QUICKSORT(array, left, last index)
22: end if
23: end procedure

The main idea of QuickSort is based on partitioning (divides) a given array A[p..r] into two parts (sub-arrays), A[p..q − 1] and A[q + 1..r], where A[q] is known as a pivot such that each element in A[p..q − 1] is less than or equal to the pivot. Consequently, all items in A[q + 1..r] will be greater than the pivot. Then, QuickSort continues (conquers) via sorting the two sub-arrays A[p..q − 1] and A[q + 1..r] by repeating the same dividing method recursively. Because the sub-arrays are already sorted, no work is needed to combine them. The whole array A[p..r] is now sorted [3].

Normally there are a few variables involved in the analysis of an algorithm. For QuickSort, three natural quantities are involved:
• the number of partitioning stages
• the number of swaps
• the number of compares

In practice, the exact values of these quantities depend on several properties of software and hardware architecture. On average, QuickSort uses [4]:

\[ \frac{n - 1}{2} \] partitioning stages

\[ 2n \ln n - 1.846n \] compares

\[ 0.333n \ln n - 0.865n \] swaps

For more detail about QuickSort analysis, see [29,32].

In this section, we explore the QuickSort sorting algorithm and its variants in the literature. Sedgewick [32] presented an improvement on QuickSort called the median-of-three modification. Chern and Hwang [51] gave an analysis of the transitional behaviors of the average cost from the insertion sort to QuickSort with the median-of-three. Fouz et al. [52] provided a smoothed analysis of Hoare’s algorithm, who designed QuickSort.

Fredman [53] presented a new and simple argument for bounding the expected running time of the QuickSort algorithm. Bindjeme and Fill [54] obtained an exact formula for the L²-distance of the (normalized) number of comparisons of QuickSort under the uniform model to its limit. Neininger [55] proved a central limit theorem for the error and obtained the asymptotics of the L3-distance. Fuchs [56] used the moment transfer approach to re-prove Neininger’s result and obtained the asymptotic of the L_p-distance for all \( 1 \leq p < \infty \).

Recently, we have encountered a number of investigations of the dual-pivot QuickSort, which is the modification of the standard QuickSort algorithm. In the partitioning step of the dual-pivot QuickSort, two pivots are used to split the sequence into three segments recursively. This approach can be made in different ways. The most efficient algorithm for the selection of the dual-pivot was developed due to the Yaroslavskiy
Question [57]. Nebel, Wild, and Martinez [58] explain the success of Yaroslavskiy’s new dual-pivot QuickSort algorithm in practice. Wild and Nebel [59] analyze this algorithm and show that on average, it uses $1.9n \ln n + O(n)$ comparisons to sort an input of size $n$, exceeding the standard QuickSort, which uses $2n \ln n + O(n)$ comparisons. Aumüller and Dietzfelbinger [60] proposed a model that includes all dual-pivot algorithms, provide a unified analysis, and identify new dual-pivot algorithms for the minimization of the average number of key comparisons among all possible algorithms. This minimum is $1.8n \ln n + O(n)$.

### 2.2 Search Algorithms

All the search algorithms in this thesis are supposed to run on a sorted array. This study is not intended algorithms that are used implicitly inside other data structures, such as the binary search tree (BST), ternary search tree in [61] or the ternary search on graphs in [62] and the interpolation search tree (IST) presented in [63,64].

#### 2.2.1 Binary Search

Although several search algorithms on the order list have been suggested in the literature, the binary search is one of the most widely-used algorithms in computer applications due to its good performance with different data types and key distributions.

The binary search works on the principle of the divide-and-conquer approach [4]. We assume that we are looking for $X$ in array $[left .. right]$ where $left$ and $right$ are position indices for the leftmost and the rightmost elements, respectively. Initially, the binary search calculates the position of the middle item of the given array using $\text{mid} = (left + right)/2$; then it compares $X$ with the middle item. If $X$ is less than the middle item, it sets $\text{right} = \text{mid} – 1$. If $X$ is greater than the middle item, it sets $\text{left} = \text{mid} + 1$. Otherwise, the binary search returns $\text{mid}$ (as the position of $X$) with a successful search. These steps are repeated recursively or iteratively until $\text{left}$ becomes equal to or greater than $\text{right}$. In this case, the binary search is terminated with an unsuccessful search.

The pseudo-code of the binary search is shown in Algorithm 3:
2.2.1.1 Binary Search Implementation Issues

“Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky,” states Donald Knuth [29]. Most of the implementation issues in the binary search are described in the literature. In this regard, five implementation errors are located and studied in [65]. The study in [66] involved a program to compute the semi-sum of two integers. In turn, this approach solved the problem of overflow that occurs in a binary search for very large arrays. Additionally, the author of [67] discussed a number of errors in the implementation of the binary search in the section titled the Challenge of Binary Search. However, all the cited works did not mention the issue we are addressing in this study.

In this thesis, we describe and analyze an unaddressed issue in the implementation of the binary search. Although this issue does not affect the correctness of the algorithm, it decreases the performance. The issue makes the binary search run in the maximum number of comparisons for each iteration. The analysis of correct and weak implementation will be discussed in Section 4.2.
2.2.2 Ternary Search and Quaternary Search

In an attempt to develop the binary search, many researchers have considered the idea that divides the list into three parts in each iteration. They called the resulting algorithm the ternary search [4,68,69]. Others split the list into four segments in each iteration, which is known as the quaternary search [70]. The ternary search divides the array into three parts in each iteration; theoretically, it has $\log_3(n)$ iterations, which are less than $\log_2(n)$ iterations in the binary search. However, for all ternary search implementations presented by the mentioned works, it is portrayed as a bad alternative for binary searches due to the cost of an iteration in the ternary search being higher than that in the binary search. On the other hand, we found that the quaternary search in [70] also had a large number of comparisons if compared with the binary search, even though it showed a lower number of iterations $\log_4(n)$. Experimentally, these algorithms showed slower performance when compared to the binary search in most cases.

In the cited works, we see that the number of comparisons in each iteration is higher than in the binary search. However, this makes the ternary search feasible only when the cost of access to a given key is far greater than the comparison process. Even the ternary search needs a lower number of iterations.

2.2.3 Interpolation Search

The interpolation search was presented by Peterson in [71] with an incorrect analysis [72]. A very detailed description of the algorithm is given along with the program implementation by Gonnet’s Ph.D. thesis in [73] and in [74]. The interpolation search average case and the lower band is $\log_2(\log_2 n)$ for a uniform distributed key only [72,74]. However, the interpolation search performs slowly for the non-uniform distributed key, and up to $O(n)$ in the worst case. In other words, the interpolation search is faster than the binary search under some conditions.

To search for the key $X$ in the sorted $Array[n]$, where $n$ the length of the array, the interpolation search assumes the following:

$Array[left]$: the minimum element in the array (at the beginning, $left = 1$).

$A[right]$: the maximum element in the array (at the beginning, $right = n$).
\[ \text{Cut index} = \text{left} + \frac{(X - \text{Array}[\text{left}])}{\text{Array}[\text{right}] - \text{Array}[\text{left}]} (\text{right} - \text{left}) \]  

(2.3)

The cut index is the interpolated position that is expected to contain \( X \), and if not equal to \( X \), the interpolation search cuts the array at this position and decides which part is to be searched based on a comparison between \( X \) and \( \text{Array}[\text{Cut index}] \).

The author of [75] clarified that two assumptions must be satisfied for an interpolation search to be practical. First, the access to a given key must be costly compared to a typical instruction, for example, in the case of external search when the array stored in external storage instead of internal memory. Second, the keys should be sorted and distributed uniformly. However, with the support of current powerful float point CPUs, the interpolation search becomes faster than the binary search even with an internal memory search.

On the other hand, the author of [27] criticized the interpolation search because it could not stand over distribution changes. In the literature, there exist a number of solutions to reduce the effect of this problem by proposing a hybrid approach that combines the binary search and interpolation search. In the next section, we will discuss that approach.

### 2.2.4 Hybrid (Interpolation-Binary) Search Approach

The similarity between the binary and interpolation searches helps researchers to suggest a hybrid approach that combines features from both algorithms. The binary and interpolation searches are both based on the divide-and-conquer technique. The dividing process depends on the manner of choosing the cut index \((1 \leq c \leq n)\), where \( n \) is the size of the list to be searched. To retrieve an element within a key equal to \( Y \), both the binary search and interpolation search compute the cut index and compare the key \( X_c \) with \( Y \). If \( X_c = Y \), the algorithm terminates with a successful search. If \( Y > X_c \), the required element is expected to exist in the subset \( X_c + 1 \ldots X_n \). Otherwise, the required element may reside in the subset \( X_1 \ldots X_c - 1 \). These steps (dividing and comparing) are repeated in each expected subset recursively until the list becomes
empty. In this case, the binary search and interpolation search are terminated with unsuccessful searches (the required key Y does not exist).

The main difference between binary search and interpolation search is the dividing method, which is defined by computing the cut index. However, the dividing methods used by the binary search and interpolation search have a valuable influence on the performance of these algorithms.

In the literature, there exists a hybrid solution presented in [76], which the authors of the presented algorithm call the Adaptive Search (AS). Their solution proved that their algorithm was faster than the first algorithm that was presented in [77]. However, in this thesis, we present a hybrid search, which is designed to work efficiently on unknown distributed ordered lists, with experimental results showing that our proposed algorithm has better performance when compared with other algorithms that use a similar approach. We made a comparative study between our proposed algorithm and the AS in Section 4.1.
CHAPTER 3

BIDIRECTIONAL CONDITIONAL INSERTION SORT (BCIS)

In this chapter, a developed in-place unstable algorithm is presented that shows fast performance in both relatively small arrays and for high rate duplicated elements arrays. The proposed algorithm, the Bidirectional Conditional Insertion Sort (BCIS), is well analyzed for the best, worst and average cases. Then, it is compared with well-known algorithms, namely the classical Insertion Sort (IS) and QuickSort.

BCIS has an average time complexity very close to $O(n^{1.5})$ for normally or uniformly distributed data. In other words, BCIS has a faster average case than IS for both relatively small and large arrays. Additionally, when it is compared with QuickSort, the experimental results for BCIS indicate a lower time complexity of between 70% and 10% within the data size range of 32-1500. Additionally, our BCIS operates faster in high rate duplicated elements arrays compared to the QuickSort, even for large arrays. Up to 10%-50% is achieved within the range of elements of from 28 to more than 3,000,000. The other advantage of the BCIS is that it can sort identical elements arrays or remain an equal part of an array in $O(n)$.

This chapter is organized as follows: Section 3.1 presents the proposed algorithm and pseudocode; Section 0 executes the proposed algorithm on a simple example array; Section 3.3 presents a detailed analysis of the complexity of the algorithm; and Section 3.4 discusses the obtained empirical results and compares them with other well-known algorithms.

3.1 The Proposed Algorithm BCIS

The standard insertion sort explained in [44,49,78] has one sorted part in the array located either on the left or right side. For each iteration, IS inserts only one item from the unsorted part into the proper place among the elements in the sorted part. This process is repeated until all the elements are sorted.
Our proposed algorithm minimizes the shifting operations caused by insertion processes using a new technique. This new technique assumes that there are two sorted parts located on the left and the right sides of the array, whereas the unsorted part is located between these two sorted parts. If the algorithm sorts ascendingly, the small elements should be inserted into the left part and the larger elements should be inserted into the right part. Logically, when the algorithm sorts in descending order, the insertion operations will be in the reverse direction. This is the idea behind the word ‘bidirectional’ in the name of the algorithm.

Unlike traditional insertion sorts, insertion items into two sorted parts have helped BCIS to be cost-effective regarding memory read/write operations. This benefit occurs because the length of the sorted part in the classical insertion sort is distributed to the two sorted parts in BCIS. The other advantage of the BCIS algorithm over the standard insertion sort is the ability to insert more than one item in their correct final positions in one sorting trip (internal loop iteration). Additionally, the inserted items will not suffer from shifting operations in subsequent sorting trips. Moreover, insertion into both sorted sides can be run in parallel to increase the algorithm performance (parallel work is out of the scope of this study).

In the case of an ascending sort, BCIS initially assumes that the leftmost item in array[1] is the left comparator (LC), where the left sorted part begins. Then it inserts each element into the left sorted part when this element is less than or equal to the LC. Correspondingly, the algorithm assumes the rightmost item at array[n] to be the right comparator (RC), which must be greater than the LC. Then, BCIS inserts each element greater than or equal to the RC into the right sorted part. However, the items that have values between the LC and RC are left in their positions during the entire sorting trip. This conditional insertion operation is repeated until all the elements are inserted into their correct positions.

If the LC and RC are already in their correct positions, there are no insertion operations that occur during the entire sort trip. Hence, the algorithm places at least two items in their final correct position for each iteration.
In the pseudo code (Algorithm 4 and 5), the BCIS algorithm is presented in a format that uses functions to increase the clarity and traceability of the algorithm. However, in statements 6 and 7, the algorithm sets two indexes. SL indicates the beginning of the Sorted Left side, and SR indicates the beginning of the Sorted Right side. Initially, SL and SR are set to indicate the leftmost element and the rightmost element, respectively. The main loop starts at statements 8 and stops when the left sorted part index (SL) reaches the right sorted part index (SR).

The selection of the LC and RC is processed by statements (9-30). To ensure the correctness of the insertion operations, the LC must be less than the RC. This condition is processed in statement (15). In the case of the LC being equal to the RC, statement (11), using the ISEQUAL function attempts to find an item not equal to the LC and replaces it with the LC. Otherwise, (if not found) all remaining elements in the unsorted part are equal. Thus, the algorithm should terminate at the statement (12).

Furthermore, this technique allows the identical elements array to sort in only O(n) time complexity. Statements (9) and (18-25) do not influence the correctness of the algorithm; in fact, these statements are added to enhance the performance of the algorithm. The advantage of these techniques will be discussed in the analysis in Section 3.3. Whereas, the variable “MinSize” in line 18 represents the minimum size that triggers this technique. For our experimental work, this variable set to 100.

The while statement in (31) is the beginning of the sort trip, and as mentioned previously, conditional insertions occur inside this loop depending on the value of the current item (CurrItem) in comparison with the values of LC and RC. Insertion operations are implemented by calling the functions INSRIGHT and INSLEFT.
Algorithm 4: BCIS (Main Body)

1: procedure BCIS(array, left, right)
2:   array is the array that required to sort
3:   left is the index of left most element in array
4:   right is the index of right most element in array
5:   array[i] is in-process element
6:   SL ← left                       \(\triangleright\) Sorted Left part index
7:   SR ← right                       \(\triangleright\) Sorted Right part index
8:   while SL < SR do
9:     SWAP(array, SR, SL + \(\frac{(SR-SL)}{2}\))
10:    if array[SL] = array[SR] then    \(\triangleright\) equality check
11:      if ISEQUAL(array, SL, SR)=-1 then
12:         return
13:      end if
14:    end if
15:    if array[SL] > array[SR] then
16:      SWAP (array, SL, SR)            \(\triangleright\) swap if LC > RC
17:    end if
18:    if (SR - SL) \(\geq\) MinSize then
19:      for i ← SL + 1 to (SR - SL)0.5 do
20:        if array[SR] < array[i] then
21:          SWAP (array, SR, i)
22:        else if array[SL] > array[i] then
23:          SWAP (array, SL, i)
24:        end if
25:      end for
26:    else
27:      i ← SL + 1
28:    end if
29:   LC ← array[SL]
30:   RC ← array[SR]
31:   while i < SR do
32:     CurrItem ← array[i]
33:     if CurrItem \(\geq\) RC then
34:       array[i] ← array[SR - 1]
35:       INSRIGHT(array, CurrItem, SR, right)
36:       SR ← SR - 1
37:     else if CurrItem \(\leq\) LC then
38:       array[i] ← array[SL + 1]
39:       INSLEFT(array, CurrItem, SL, left)
40:       SL ← SL + 1
41:       i ← i + 1
42:     else
43:       i ← i + 1
44:     end if
45:   end while
46:   SL ← SL + 1
47:   SR ← SR - 1
48: end procedure
Algorithm 5: BCIS (Functions)
1: function ISEQUAL(array, SL, SR)
2:     for k ← SL + 1 to SR − 1 do
3:         if array[k]! = array[SL] then
4:             SWAP(array, k, SL)
5:             return k
6:     end if
7:  end for
8:  return −1
9: end function

1: function INSRIGHT(array, CurrItem, SR, right)
2:     j ← SR
3:     while j ≤ right and CurrItem > array[j] do
4:         array[j−1] ← array[j]
5:         j ← j+1
6:     end while
7:     Array[j−1] ← CurrItem
8: end function

1: function INSLEFT(array, CurrItem, SL, left)
2:     j ← SL
3:     while j ≥ left and CurrItem < array[j] do
4:         array[j+1] ← array[j]
5:         j ← j−1
6:     end while
7:     Array[j+1] ← CurrItem
8: end function

1: function SWAP(array, i, j)
2:     Temp ← array[i]
3:     array[i] ← array[j]
4:     array[j] ← Temp
5: end function

3.2 BCIS Example

The behavior of the proposed algorithm on an array of 15 elements generated randomly by computer is explained in Figure 3.1. To increase the simplicity of this example, we assume that statements (9) and (18-25) do not exist in the algorithm. For all examples in this chapter, we assumed the following: Items in red mean that they are currently in process. Bolded items represent LC and RC for the current sort trip. A gray background
means the position of these elements may change during the current sort trip. Finally, items with a green background mean that they are in their correct final positions.

First sort trip starts here

```
  22  17  56  57  52  59  80  78  73  19  53  28  65  72  67
```

- Insert into the left
- No insertion
- No insertion
- No insertion
- No insertion
- Insert into the right
- Insert into the right
- No insertion
- No insertion
- Insert into the right
- No insertion
- No insertion
- Insert into the right
- No insertion
- No insertion
- No insertion
- Insert into the left
- Check LC and RC, swap.

End of first sort trip.

```
  17  19  22  56  57  52  59  65  28  53  56  67  72  73  78  80
```

Second sort trip starts here.

```
  17  19  22  56  52  59  65  28  53  57  67  72  73  78  80
```

- Insert into the left
- Insert into the right
- Insert into the left
- Insert into the right
- Insert into the right
- Insert into the right
- Insert into the right
- Insert into the right
- Insert into the left
- Check LC and RC, swap.

End of second sort trip, all items has been sorted

```
  17  19  22  28  53  56  57  59  65  67  72  73  78  80
```

**Figure 3.1** BCIS example
3.3 Analysis of the Proposed Algorithm

The complexity of the proposed algorithm mainly depends on the complexity of the insertion functions, which in turn depends on the number of inserted elements in each function during each sorting trip. To explain how the performance of the BCIS depends on the number of inserted elements per sort trip, several assumptions are presented which reveal that the theoretical analysis is very close to the experimental results that we obtained.

In order to simplify the analysis, we will concentrate on the main parts of the algorithm. We assume that during each sort trip \((k)\) elements is inserted into both sides, and each side receives \(k/2\), where the insertion functions work identically to the standard insertion sort. Consequently, the time complexity of each sort trip is equal to the sum of the left and right insertion function cost, which is equal to \(T_is(k/2)\) for each function, in addition to the cost of scanning of the remaining elements (not the inserted elements). We can express this idea as follows:

\[
T(n) = T_{is}(\frac{k}{2}) + T_{is}(\frac{k}{2}) + 2(n - k) + T_{is}(\frac{k}{2}) + T_{is}(\frac{k}{2}) + 2(n - 2k) + T_{is}(\frac{k}{2}) + T_{is}(\frac{k}{2}) + 2(n - 3k) + \cdots + T_{is}(\frac{k}{2}) + T_{is}(\frac{k}{2}) + 2(n - ik)
\]

The BCIS stops when \((n - ik) = 0 \Rightarrow i = \frac{n}{k}\)

\[
T(n) = \frac{n}{k} \left[ T_{is}\left(\frac{k}{2}\right) + T_{is}\left(\frac{k}{2}\right) \right] + \sum_{i=1}^{\frac{n}{k}} (n - ik)
\]

\[
= \frac{n}{k} \left[ T_{is}\left(\frac{k}{2}\right) + T_{is}\left(\frac{k}{2}\right) \right] + \frac{n^2}{k} - n
\]
\[ \frac{n}{k} \left[ T_{is} \left( \frac{k}{2} \right) + T_{is} \left( \frac{k}{2} \right) + n \right] - n \]  \hspace{1cm} (3.2)

Eq. (3.2) represents a general form of the growth function. It shows that the complexity of the proposed algorithm mainly depends on the value of \( k \) and the complexity of the insertion functions.

### 3.3.1 Average Case Analysis

The average case of the classical insertion sort \( T_{is}(n) \) that appears in Eq. (3.2) has been well analyzed in terms of comparisons in [36,37] and, and in [36] for assignments. However, the authors of the cited works presented Eqs.(2.1) and (2.2) (discussed in Section 2.1.1), which represent the average case analysis of classical insertion sort for comparisons and assignments, respectively.

Eqs. (2.1) and (2.2) show that the insertion sort has approximately an equal number of comparisons \( (T_{isc}(n)) \) and assignments \( (T_{isa}(n)) \).

However, for the BCIS, it is assumed that in each sort trip, \( k \) elements are inserted into both sides. Therefore, the main while loop executes \( n/k \) times, which represents the number of sort trips. We can assume that each insertion function receives \( k/2 \) elements where \( 2 \leq k \leq n \). Since both insertion functions (INSLEFT, INSRIGHT) work exactly as a standard insertion sort, the average case of comparisons for each function during each sort trip is:

\[
\text{Comp. # / Sort Trip / Function} = T_{isc} \left( \frac{k}{2} \right)
\]

\[
= \frac{k^2}{16} + \frac{3k}{8} - 1
\]  \hspace{1cm} (3.3)

The BCIS performs one extra assignment operation to move the element that neighbored the sorted section before calling each insertion function. By considering this cost, we obtained the following:
Assig. # / Sort Trip / Function = \( T_{\text{lsa}} = \left( \frac{k}{2} \right) + \frac{k}{2} \)

\[
= \frac{k^2}{16} + \frac{11k}{8} + 3
\]  

(3.4)

In order to compute the BCIS comparisons complexity, we substituted Eq. (3.3) into Eq. (3.2), and we obtained:

\[
T_c(n) = \frac{n}{k} \left[ \frac{k^2}{8} + \frac{3k}{4} - 2 + n \right] - n
\]

(3.5)

Eq. (3.5) shows that when \( k \) receives a small value, the algorithm performance approaches close to \( O(n^2) \). For \( k = 2 \), the growth function becomes:

\[
T_c(n) = \frac{n}{2} \left[ \frac{4}{8} + \frac{3}{4} - 2 + n \right] - n
\]

\[ T_c(n) = \frac{n^2}{2} - 1.375 n \]  

(3.6)

When \( k \) has a large value, the complexity of the BCIS also approaches close to \( O(n^2) \). For \( k = n \), the complexity is:

\[
T_c(n) = \frac{n}{n} \left[ \frac{n^2}{8} + \frac{3n}{4} - 2 + n \right] - n
\]

\[ T_c(n) = \frac{n^2}{8} + \frac{3n}{4} - 2 \]  

(3.7)

Hence, the best performance of the average case for the BCIS that could be obtained was when \( k = n^{0.5} \), as follows:
Eq. (3.8) computes the number of comparisons of the BCIS when it runs in the best performance of the average case. On the other hand, it calculates the number of assignments for the BCIS in the case of the best performance of average cases. Since assignments operations occur only in insertions functions, Eq. (3.4) is multiplied by 2 because there are two insertion functions; then the result is multiplied by the number of the sort trip $n/k$. When $k = n^{0.5}$, we produce the following:

$$T_a(n) = \frac{n}{n^{0.5}} + 11n^{0.5} + 6$$

$$T_a(n) = \frac{n^{1.5}}{8} + 11n^{0.5} + 6n^{0.5} \tag{3.9}$$

The comparison of Eq. (3.8) with Eq. (3.9) proves that the number of assignments is lower than the number of comparisons in the BCIS. As mentioned previously in Eqs. (2.1) and (2.1), the IS has approximately an equal number of comparisons and assignments. This property makes the BCIS run faster than the IS even when they have a similar number of comparisons.

Hence, we wrote the code section in statements (18-25) to optimize the performance of the BCIS by keeping $k$ as close as possible to $n^{0.5}$. This code segment was based on the idea that ensured at least a set of length $(SR - SL)^{0.5}$ not be inserted during the current sort trip (where the sort trip size = $SR - SL$). This idea was realized by scanning this set looking for the minimum and maximum elements and replacing them with LC and RC, respectively. However, this code does not add extra cost for the performance of the algorithm because the current sort trip will start where the loop in statement (19) has finished (the sort trip will start at $(SR - SL)^{0.5} + 1$). The theoretical
results were compared with the experimental results in Section 3.4 and the BCIS showed a performance close to the best performance of the average case explained above.

In Algorithm 6, the instruction level analysis of the BCIS is presented. We re-analyze the algorithm for the average case by applying the above assumption to obtain a more detailed analysis. However, the cost of each instruction is demonstrated by a comment in the pseudo code of the algorithm. We do not explicitly calculate the cost of the loop in statement (19) because it is implicitly calculated with the cost of not inserting elements inside the loop starting with statement (30). The code segment within statements (18-25) activates for sort trip sizes greater than 100 elements only. Otherwise, sort trip index $i$ starts from the element next to SL (statement 27).

The total number of comparisons for each insertion function is calculated with Eq. (3.8) multiplied by the number of sort trips ($n/k$), as follows:

$$\frac{n}{k} \left( \frac{k^2}{16} + \frac{3k}{8} - 1 \right) = \frac{nk}{16} + \frac{3n}{8} - \frac{n}{k}$$ (3.10)

The complexity of the check equality function ISEQUAL is disregarded because the \textit{if} statement at (10) is rarely true.
Algorithm 6: BCIS Average case analysis

1: procedure BCIS(array, left, right)   
2:     array is the array that required to sort   
3:     left is the index of left most element in array   
4:     right is the index of right most element in array   
5:     SL ← left   
6:     SR ← right   
7:     while SL < SR do   
8:         SWAP(array, SR, SL + (SR - SL) \(\frac{2}{k}\))   
9:         if array[SL] = array[SR] then   
10:             if ISDUP(array, SL, SR)=1 then   
11:                 return   
12:         end if   
13:         if array[SL] > array[SR] then   
14:             SWAP (array, SL, SR)   
15:         end if   
16:         if (SR - SL) \(\geq\) MinSize then   
17:             for i ← SL + 1 to (SR - SL) \(0.5\) do   
18:                 if array[SR] < array[i] then   
19:                     SWAP (array, SR, i)   
20:                 end if   
21:             end for   
22:         else   
23:             i ← SL + 1   
24:         end if   
25:     end while   
26:     LC ← array[SL]   
27:     RC ← array[SR]   
28:     while i < SR do   
29:         CurItem ← array[i]   
30:         if CurItem ≥ RC then   
31:             array[i] ← array[SR - 1]   
32:             INSRIGHT(array, CurItem, SR, right)   
33:             SR ← SR - 1   
34:         else if CurItem ≤ LC then   
35:             array[i] ← array[SL + 1]   
36:             INSLEFT(array, CurItem, SL, left)   
37:             SL ← SL + 1   
38:             i ← i + 1   
39:         else   
40:             i ← i + 1   
41:         end if   
42:     end while   
43:     SL ← SL + 1   
44:     SR ← SR - 1   
45: end procedure
The total complexity of the BCIS is calculated as follows:

\[ T(n) = C1 + C2 + C3 + (C3 + C4 + C5 + C6 + C7 + C8 + C21 + C22) \frac{n}{k} \]

\[ + (C10 + C11 + C12 + C16 + C20) \sum_{i=1}^{n} (n - ik) \]

\[ + (C13 + C15 + C16 + C17 + C18 + C19) \frac{n}{2} \]

\[ + C14 \times 2 (\frac{nk}{16} + \frac{3n}{8} - \frac{n}{k}) - C20n \]

where we assume:

\[ a = (C3 + C4 + C5 + C6 + C7 + C8 + C21 + C22) \]

\[ b = C14 \]

\[ c = (C10 + C11 + C12 + C16 + C20) \]

\[ d = (C13 + C15 + C16 + C17 + C18 + C19) \]

\[ e = C20 \]

\[ f = C1 + C2 + C3 \]

\[ T(n) = a \times \frac{n}{k} + b (\frac{nk}{8} + \frac{3n}{4} - \frac{2n}{k}) + c \sum_{i=1}^{n} (n - ik) + d \times \frac{n}{2} - en + f \]

\[ = a \times \frac{n}{k} + b (\frac{nk}{8} + \frac{3n}{4} - \frac{2n}{k}) + c \left( \frac{n^2}{2k} - \frac{n}{2} \right) + d \times \frac{n}{2} - en + f \]
\[ T(n) = \frac{n}{k} \left[ a + b \left( \frac{k^2}{8} + \frac{3k}{4} - 2 \right) + c \frac{n}{2} \right] - c \frac{n}{2} + d \frac{n}{2} - en + f \] (3.11)

We notice that Eq. (3.11) is similar to Eq. (3.5), whereas the constants represent instructions cost.

### 3.3.2 Best Case Analysis

The best-case complexity of the BCIS occurs in situations of every element being placed in its correct final position consuming a limited and constant number of comparisons and shift operations in one sort trip. These conditions are available once the first sort trip starts while the RC and LC are holding the largest and second largest item in the array, respectively, and all other elements are already sorted. The following example in Figure 3.2 explains this best case (element 15 will be replaced with 7 in statement 9).

![Figure 3.2 Best case example for less than 100 elements](image)

For this best case, we note that all insertions will be on the left side only with one shifting operation per insertion. This means that the cost of the insertion of each item is O(1). Therefore, the total cost of the left insertion function is \( T_{ls}(n) = n \). Moreover, all elements will have been inserted in one sort trip, so \( k = n \). These values are substituted in Eq. (3.1), as follows:

\[ T(n) = \frac{n}{k} \left[ T_{ls}(n) \right] + \sum_{i=1}^{n} (n - ik) \]

where \( k = n \)

\[ T(n) = n \] (3.12)
Hence, the best case of the BCIS is \( O(n) \) for \( n < 100 \). Otherwise, for \( n \geq 100 \), the loop starts with statement (19), which always prevents this best case from occurring because it only puts LC and RC into their correct positions and disallows insertions during all sort trips. As a result, the loop in statement (19) forces the algorithm to run very slow on already sorted or reverse sorted arrays.

Generally, the already sorted and reverse sorted arrays are more common in practice if compared with the above best case example. Therefore, statement (9) has been added to enhance the performance of the best case and the worst case when the BCIS runs on ordered and reverse ordered arrays. In the case of an already sorted array, this statement makes the BCIS, during each sort trip, insert half of \((SR - SL)\) at the least cost.

The example in Figure 3.3 explains how the BCIS runs on an already sorted array. For simplicity, elements not inserted are not represented in each sort trip during the demonstration of this example.

![Figure 3.3 Example of running BCIS on an already sorted array](image-url)
For an already sorted array, the BCIS scans the first half consuming two comparisons per item (no insertions), followed by inserting the second half of each sort trip and consuming two comparisons per item. Therefore, the sort trip size is repeatedly halved. This can be represented as follows:

\[
T(n) = \left(\frac{2n}{2} + 2 \frac{n}{2}\right) + \left(\frac{2n}{4} + 2 \frac{n}{4}\right) + \left(\frac{2n}{8} + 2 \frac{n}{8}\right) + \cdots + \left(\frac{2n}{2^i} + 2 \frac{n}{2^i}\right)
\]

Stops when \(2^i = n \Rightarrow i = \log(n)\)

\[
T(n) = \sum_{i=1}^{\log n} \left(\frac{2n}{2^i} + 2 \frac{n}{2^i}\right) = 4n \quad (3.13)
\]

Eqs. (3.12) and (3.13) represent the best-case growth functions of the BCIS when running on array sizes less than 100 and greater than or equal to 100, respectively.

#### 3.3.3 Worst Case Analysis

The worst case occurs only if all elements are inserted in reverse on one side during the first sort trip. This condition is provided when the RC and LC are respectively the largest and second largest numbers in the array, and all other items are sorted in reverse order. The insertion occurs on the left side only. The following example in Figure 3.4 explains this worst case when \(n < 100\).

```
14 13 12 11 10 9 8 15 6 5 4 3 2 1 7
```

**Figure 3.4** Worst case example for arrays less than 100 elements

Since each element in this example is inserted in reverse, the complexity of the left insertion function for each sort trip is equal to \(T_i(n) = n(n - 1)/2\). Moreover, there is one sort trip, so \(k = n\); by substituting these values into Eq.(3.2), we have:

\[
T(n) = \frac{n}{k} [T_i(n)] + \sum_{i=1}^{n/k} (n - ik)
\]
where \( k = n \)

\[
T(n) = \frac{n(n - 1)}{2}
\]  
(3.14)

Hence, the worst case of the BCIS is \( O(n^2) \) for \( n < 100 \). Similarly, for the situation in the best case, the loop in statement (19) prevents the worst case from happening because LC will not take the second largest item in the array. Consequently, the worst case for the BCIS would be when it runs on the reverse sorted array for \( n \geq 100 \). The example in Figure 3.5 explains the behavior of the BCIS on such arrays even if the array size is less than 100.

In the case of a reverse sorted array, the BCIS does not insert the first half of the scanned elements, costing two comparisons per each item. It then inserts the second half in reverse for each sort trip. The cost of a reverse insertion (for each sort trip) is

![Figure 3.5 Example of running BCIS on a reverse sorted array](image-url)
$T_{\text{BCIS}}(k) = k(k - 1)/2$ where $k$ is halved repeatedly. Similarly to the already sorted array analysis, the complexity of the BCIS can be represented as follows:

$$T(n) = \left(2 \frac{n}{2} + \frac{(n/2)^2 - n/2}{2}\right) + \left(2 \frac{n}{4} + \frac{(n/4)^2 - n/4}{2}\right) + \left(2 \frac{n}{8} + \frac{(n/8)^2 - n/8}{2}\right) + \cdots$$

$$+ \left(2 \frac{n}{2^i} + \frac{(n/2^i)^2 - n/2^i}{2}\right)$$

Stops when $2^i = n \Rightarrow i = \log n$

$$T(n) = \sum_{i=1}^{i=\log n} \left(2 \frac{n}{2^i} + \frac{(n/2^i)^2 - n/2^i}{2}\right)$$

$$T(n) = \frac{n^2}{6} + \frac{3n}{2}$$  (3.15)

Eqs. (3.14) and (3.15) represent the worst-case growth functions of the BCIS when running on array sizes of less than 100 and greater than and equal to 100.

### 3.4 Results and Comparison with Other Algorithms

The proposed algorithm is implemented in C++ using NetBeans 8.0.2 IDE based on the Cygwin compiler. The measurements are taken on a 2.1 GHz Intel Core i7 processor with 6 GB 1067 MHz DDR3 memory machine on the Windows platform.

The experimental test was conducted on empirical data (numbers) that were generated randomly using a C++ library. These library classes generated specified ranges of random numbers with different distributions [79]. The real-world data that were used in this study were downloaded from [80]. All results comparisons used the Uniform Distribution except for those shown in Figure 3.9.
3.4.1 BCIS and classical insertion sort

Figure 3.6 explains the average number of comparisons and assignments (Y-axis) for the BCIS and IS with different list sizes (X-axis). The figure was plotted using Eqs. (2.1), (2.2), (3.8), and (3.9). This figure explains that the number of comparisons and assignments of the IS are approximately equal. In contrast, in the BCIS the number of assignments is lower than the number of comparisons. This feature shows the better performance of the BCIS and supports our claim that the BCIS has fewer memory read/write operations when compared with the IS.

The average theoretical analysis for the BCIS and IS is calculated in terms of the number of comparisons and assignments separately in Eqs. (2.1), (2.2), (3.8), and (3.9). To compare the results of these equations with experimental results of the BCIS and IS, which are measured by the execution-time (elapsed running time of an algorithm), we represent these quantities as a theoretical and experimental ratio of $\frac{BCIS}{IS}$.
In this theoretical ratio, we assumed that the cost of a comparison operation and assignment is equal in both algorithms. Therefore, Eq. (2.1) was added to Eq. (2.2) in order to compute the total cost of the IS (IS\textsubscript{total}). Similarly, the total cost of the BCIS (BCIS\textsubscript{total}) is the result of adding Eq. (3.8) to Eq. (3.9).

Figure 3.7 illustrates a comparison in performances of the proposed BCIS algorithm and the IS algorithm. This comparison has been represented as the ratio \( \frac{BCIS}{IS} \) (Y-axis) which is required to sort a list of random data for a number of list sizes (X-axis). Theoretically, this ratio is equal to \( \frac{BCIS\textsubscript{total}}{IS\textsubscript{total}} \). However, in practice, the ratio is computed experimentally by dividing BCIS\textsubscript{time} by IS\textsubscript{time}, where these parameters represent the experimental elapsed running times of the BCIS and IS, respectively.

In the experimental \( \frac{BCIS\textsubscript{time}}{IS\textsubscript{time}} \) ratio, we noticed that the proposed algorithm has roughly equal performance when compared to the classical insertion sort for list sizes less than 50 (\( \frac{BCIS\textsubscript{time}}{IS\textsubscript{time}} = 1 \)). However, the performance gain of the BCIS increased noticeably
for larger list sizes. The time required to sort the same size of list using the BCIS begins at 70%, which then declines to 4% of that consumed by the classical insertion sort for list sizes up to 6000.

Figure 3.8 explains the same ratio for $n > 6000$. This figure shows that $\frac{\text{BCIS\,time}}{\text{IS\,time}}$ decreases when the list size increases. For instance, for a list size of 3,643,076, the experimental ratio is equal to 0.00128, which means that the BCIS is 781 times faster than the IS.

![Figure 3.8 BCIS/IS ratio for $n > 6000$](image)

In conclusion, Figure 3.7 and Figure 3.8 show that the theoretical and experimental ratios are very close, especially for large lists. This means that the BCIS approaches the best performance of the average case for large datasets.

The experimental BCIS/IS for several distributions has been tested and compared with real-world data in Figure 3.9. This figure shows that the BCIS has approximately the same performance on the Uniform, Normal and Poisson distributions as well as the
real-world data. However, the binomial distribution class in [79] has been configured to generate more duplicated numbers compared with other distributions to show the effect of a moderate rate of duplicate elements. For this reason, the binomial distribution shows slightly better performance compared with other distributions. This occurs because duplicated elements reduce the comparisons and assignments (shifting operations) inside inserting functions in the BCIS.

![Figure 3.9 BCIS/IS ratios with different data distributions](image)

The BCIS concisely shows approximately equal performance for all tested distributions and real-world data with low or no duplicated elements, while the performance of the BCIS increases when duplicated elements increase. For high rates of duplicated elements, the BCIS is compared with QuickSort in Section 3.4.2.2.
3.4.2 BCIS and QuickSort

3.4.2.1 BCIS and QuickSort Comparison for No Duplicated-Elements Dataset

Figure 3.10 presents a comparison in experimental performance of the proposed BCIS algorithm and QuickSort for small lists.

A widely used enhanced version of QuickSort (median-of-three pivots) is used, which is suggested by [32]. This comparison is represented in terms of the experimental ratio $\frac{\text{BCIS time}}{\text{QuickSort time}}$ (Y-axis) that is required to sort a list with random data for a number of list sizes (X-axis). We can observe that the BCIS is faster than QuickSort for list sizes less than 1500 in most cases. The time required to sort the same size of lists using the BCIS ranged between 30% and 90% of that consumed by QuickSort when list sizes were less than 1500.

![Experimental BCIS/QuickSort](chart.png)

**Figure 3.10** BCIS and QuickSort performance $n < 2000$

Although theoretical analysis in all previously cited works that have been explained in the literature (Section 2.1.2) showed that QuickSort has more efficient comparison...
complexity than BCIS, experimentally the BCIS outperforms QuickSort for relatively small array sizes for a number of reasons, including the low number of assignment operations in BCIS, and an assignment operation having a lower cost if compared with the swap operation used in QuickSort. Each swap operation requires three assignments to perform. Finally, due to the nature of the cache memory architecture [81], the BCIS uses cache memory more efficiently because shifting operations only have access to adjacent memory locations, while the swapping operations in QuickSort access memory randomly. Therefore, QuickSort cannot efficiently invest the remarkable speed gain that is provided by the cache memory. Figure 3.11 shows the experimental $\frac{\text{BCIS time}}{\text{QuickSort time}}$ ratio for array sizes greater than 2000 and less than 3,640,000.

![Figure 3.11 Experimental BCIS/QuickSort ratio for $n > 2000$.](image)
3.4.2.2 BCIS and QuickSort Comparison for High Rates of Duplicated Elements Dataset

The experimental test showed that the BCIS is faster than QuickSort when running on data sets with a high rate of duplicated elements, even for large list sizes. Figure 3.12 explains the experimental ratio BCIS/QuickSort when the array being used has only 50 different elements. The computer randomly duplicates the same 50 elements for arrays of sizes greater than 50. This figure shows that the BCIS consumes only 50% to 90% of the time consumed by QuickSort when being run on the high rate of duplicated elements array. This variation in the ratio is due to the random selection of LC and RC during each sort trip. The main factor that makes the BCIS faster than QuickSort for these types of array is that there is a small number of assignments and comparisons operations, and if there are many numbers equal to LC or RC in each sort trip. This case has a high probability of occurring when there is a high rate of duplicated elements in the array.

![Experimental BCIS/QuickSort ratio for 50 duplicated elements](image)

*Figure 3.12 Experimental BCIS/QuickSort ratio for 50 duplicated elements*
CHAPTER 4

DEVELOPMENT SEARCH ALGORITHMS FOR ORDERED DATASET

In this chapter, we present a hybrid algorithm to search ordered datasets. The presented algorithm is called the Hybrid Search (HS). Our proposed algorithm was designed for a dataset with an unknown distribution of keys. It experimentally shows a good performance for Uniform and Non-Uniform distributed keys. The HS combines the two dividing techniques that are used in interpolation and binary searches.

Additionally, we present a precise method to analyze the binary search in terms of comparisons, and we highlight and analyze a common error in the implementation of the binary search. Then, we present two efficient search algorithms, the first of which is an improved version of the ternary search, and the second being a new algorithm using a new binary/quaternary dividing technique. Both proposed algorithms experimentally show better performance when compared with the binary search.

4.1 The Proposed Hybrid Search (HS) Algorithm

The HS is designed to run competently in both uniform and non-uniform distributed sorted arrays. It combines the basics of the binary and interpolation searches efficiently to take advantage of both algorithms.

We assume that the HS looks to $X$ among the sorted list $Array [\text{Left} .. \text{Right}]$. During each iteration, the HS initially calculates the estimated position by using the standard interpolation search method, followed by running a modified version of the binary search on the remaining part of the array which is expected to hold the required key.

Algorithm 7 provides the pseudo code for the HS. By looking at the pseudo code more closely, we find that the HS firstly uses the interpolation technique to estimate the location of the searched key.
The calculated index is stored in the variable \texttt{Inter} (line no. 7). Then the HS compares the required key \( X \) with the content of the estimated location \texttt{Array[Inter]} in (lines no. 8 and 16) to decide whether the required key is resident on the left or right segment with respect to \texttt{Inter}. If \( X \) is located on the left or the right part, the HS divides this

\begin{algorithm}
\textbf{Algorithm 7: Hybrid Search (HS)}

1: \textbf{procedure} HS( \texttt{array, left, right, X})
2: \hspace{1em} \texttt{array} is the array that \textbf{required} to search
3: \hspace{1em} \texttt{left} is the index of left most element in \texttt{array}
4: \hspace{1em} \texttt{right} is the index of right most element in \texttt{array}
5: \hspace{1em} \texttt{X} is the element that we search for
6: \hspace{1em} \textbf{while} \texttt{left} < \texttt{right} \textbf{do} \hspace{1em} \triangleright 1 \text{ comparisons}
7: \hspace{2em} \texttt{Inter} \leftarrow \texttt{left} + \frac{(X - \texttt{array[left]}) \cdot (\texttt{right} - \texttt{left})}{(\texttt{array[right]} - \texttt{array[left]})}

8: \hspace{2em} \textbf{if} \texttt{X} > \texttt{array[Inter]} \textbf{then} \hspace{1em} \triangleright \text{ to check the right portion of Inter}
9: \hspace{3em} \texttt{Mid} \leftarrow \frac{\texttt{Inter} + \texttt{right}}{2} \hspace{1em} \triangleright 2 \text{ comparisons}
10: \hspace{3em} \textbf{if} \texttt{X} \leq \texttt{array[Mid]} \textbf{then}
11: \hspace{4em} \texttt{left} \leftarrow \texttt{Inter} + 1 \hspace{1em} \triangleright 3 \text{ comparisons}
12: \hspace{4em} \texttt{right} \leftarrow \texttt{Mid}
13: \hspace{3em} \textbf{else}
14: \hspace{4em} \texttt{left} \leftarrow \texttt{Mid} + 1
15: \hspace{3em} \textbf{end if}
16: \hspace{2em} \textbf{else if} \texttt{X} < \texttt{array[Inter]} \textbf{then} \hspace{1em} \triangleright \text{ to check the left portion of Inter}
17: \hspace{3em} \texttt{Mid} \leftarrow \frac{\texttt{Inter} + \texttt{left}}{2} \hspace{1em} \triangleright 3 \text{ comparisons}
18: \hspace{3em} \textbf{if} \texttt{X} \geq \texttt{array[Mid]} \textbf{then}
19: \hspace{4em} \texttt{left} \leftarrow \texttt{Mid}
20: \hspace{4em} \texttt{right} \leftarrow \texttt{Inter} - 1 \hspace{1em} \triangleright 4 \text{ comparisons}
21: \hspace{3em} \textbf{else}
22: \hspace{4em} \texttt{right} \leftarrow \texttt{Mid} - 1
23: \hspace{3em} \textbf{end if}
24: \hspace{2em} \textbf{else}
25: \hspace{3em} \textbf{return} \texttt{Inter}
26: \hspace{3em} \textbf{end if}
27: \hspace{2em} \textbf{end while}
28: \hspace{1em} \textbf{if} \texttt{X} = \texttt{array[left]} \textbf{then}
29: \hspace{2em} \textbf{return} \texttt{left}
30: \hspace{1em} \textbf{end if}
31: \hspace{1em} \textbf{return} -1 \hspace{1em} \triangleright \text{ not found}
32: \textbf{end procedure}
\end{algorithm}
segment into two halves by calculating the \textbf{Mid} variable at (lines no. 9 and 17). Hence, the HS minimizes the search space by setting new values to the \textbf{Right} and \textbf{Left} variables. Following this process, the new search space for the next iteration is minimized into half of the interpolated portion that was calculated by an interpolation method. This approach minimizes the total number of iterations in all tested distributions. However, the HS returns the location of the searched key in (lines 25 and 29). Line 28 is written to catch a case where the loop terminates and the element at \textbf{Array[Left]} is not checked (for more detail, follow the example in Table 4.1. In the case of the HS reaching line no. 31, the algorithm ends with an unsuccessful search.

Figure 4.1 visually shows the division technique used by the HS. We assume that the HS estimates the required key at index \textbf{Inter} (in Figure 4.1). If \textbf{X} is greater than \textbf{Array[Inter]}, the HS calculates the midpoint (\textbf{Mid}) of the range from \textbf{Inter} to \textbf{Right}. Then it selects one of these halves according to the value of \textbf{X}. Similarly, the HS follows the same procedures for the left portion ranging from \textbf{Left} to \textbf{Inter} when \textbf{X} is less than \textbf{Array[Inter]}.

\textbf{Figure 4.1} Dividing technique of the HS

\subsection{4.1.1 Complexity of Hybrid Search}

Although the HS has higher iteration costs when compared with the binary and interpolation searches due to its computations, the HS showed fewer iterations when compared with the binary search for all the tested distributions and fewer iterations than the interpolation search iterations for non-uniform distributions.

When the HS runs on uniform distributed keys, the average case complexity of the HS is equal to the complexity of the interpolation search, which is $O(\log_2 (\log_2 n))$. This is due to the HS starting with the interpolation division technique in each iteration.
In the case of non-uniform distributed keys, the interpolation division technique works in conjunction with the binary division technique to reduce the number of iterations. However, HS worst case complexity is still $O(\log_2 n)$ in non-uniform distributions because the HS divides the interpolated segment into two halves in each iteration.

The experimental results explain that the number of iterations of the HS is, on average, less than half of the number of iterations of the binary search. Consequently, the HS shows execution times that are far lower than those of the interpolation search and are very close to those of the binary search for certain array sizes.

### 4.1.2 Hybrid Search Example

In this example, we randomly generated an ordered array that has a location index starting from 0 to 34 with random normally distributed keys. Then we searched all the keys in this array using the HS. Table 4.1 presents the calculated variables (Inter and Mid) during each iteration in the HS. The bolded numbers in the table represent the index of the returned location by line no. 25 (return Inter) or line no. 28 (return Left) in the pseudo code section.

The HS example given in Table 4.1 shows iterations numbered between 1 and 3. The average iteration number equals 2.31, while the average number of iterations for the binary search on the same array equals 4.73, which is less than $\log_2(35) = 5.12$.

By looking at Table 4.1, we see the HS found the first key and the last key (at indices 0 and 34) in one iteration only. This was due to the nature of the interpolation dividing technique that is used by the HS. Based on this reality, we assume we are seeking the item in the middle of the list (at index 17 in the example). The HS certainly finds the middle item on the second iteration because this item becomes the last item after the first iteration.

However, during each dividing process, Mid indexes to the one end of the new list and the other end indexed by left or Inter or right (see Figure 4.1). This method makes the HS find the required item exactly at the next iteration if it is located at the end of the segment. Moreover, there is a higher probability of finding the searched item in
the next iteration if it is located near to one end. Thus, this method enhances the performance of the HS, especially for small or moderately sized arrays.

Table 4.1 Example of running HS on normally distributed array

<table>
<thead>
<tr>
<th>Index</th>
<th>Element/Key</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Inter</td>
<td>Mid</td>
<td>Inter</td>
</tr>
<tr>
<td>0</td>
<td>6.983</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8.954</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11.74</td>
<td>11</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>11.774</td>
<td>11</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>12.192</td>
<td>12</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>12.316</td>
<td>12</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>12.682</td>
<td>13</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>12.752</td>
<td>13</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>12.939</td>
<td>14</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>12.962</td>
<td>14</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>13.42</td>
<td>15</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>13.435</td>
<td>15</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>13.99</td>
<td>16</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13.994</td>
<td>16</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>14.446</td>
<td>17</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>14.616</td>
<td>18</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>14.634</td>
<td>18</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>14.779</td>
<td>18</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>14.961</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>15.1</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>15.101</td>
<td>19</td>
<td>26</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>15.151</td>
<td>19</td>
<td>26</td>
<td>20</td>
</tr>
<tr>
<td>22</td>
<td>15.602</td>
<td>20</td>
<td>27</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>16.115</td>
<td>21</td>
<td>27</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>16.131</td>
<td>21</td>
<td>27</td>
<td>23</td>
</tr>
<tr>
<td>25</td>
<td>16.388</td>
<td>22</td>
<td>28</td>
<td>24</td>
</tr>
<tr>
<td>26</td>
<td>17.053</td>
<td>23</td>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>27</td>
<td>17.235</td>
<td>24</td>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>28</td>
<td>17.35</td>
<td>24</td>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>17.418</td>
<td>24</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>17.575</td>
<td>25</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>18.063</td>
<td>26</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>18.925</td>
<td>28</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>19.207</td>
<td>28</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>34</td>
<td>21.374</td>
<td>34</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.1.3 Hybrid Search and Adaptive Search Comparisons

Fortunately, some authors have provided the source code of the Adaptive Search (AS) in [82]. The Adaptive Search is the algorithm presented in [76]. It was originally written in Java; we simply converted it to C++ code without altering the behavior of the algorithm so as to fit our test requirements. Although the HS and AS revealed close iteration numbers in most cases, the HS remarkably has lower iteration costs in terms of comparison numbers and computation.

Algorithm 8 represents the pseudo code of AS. AS initially calculates the values of mid as the middle point and next using the interpolation technique. Then, regarding the interpolated position (next), the AS divides the selected portion (left or right segment) using the binary search technique if, and only if, one of the following two conditions is true (the first two if statements are in AS):

\[
\text{if } ((\text{key} < a[\text{next}]) \text{ and } ((\text{next} - \text{left}) > \text{mid}))
\]

\[
\text{if } ((\text{key} > a[\text{next}]) \text{ and } ((\text{right} - \text{next}) > \text{mid}))
\]

otherwise, AS continues as a normal interpolation search.

The main flaw in the dividing technique used in the AS is that it may consume four comparisons operations (inside the two mentioned if statements) without dividing the given segments into two halves. In this regard, there is a probability of doing extra work in some iterations.

For instance, we assume we are using the AS to search for the element at index 2 in Table 4.1. In the first iteration bot = 0 and top = 34, the AS calculates mid = 17 and next = 11. In this case, both if statements are not true. However, the probability of occurrence of this case increases with a non-uniform distribution.
The advantages of our proposed algorithm over the AS are that our algorithm uses both the dividing techniques of the binary and interpolation searches in each iteration using a lower number of comparisons. In the pseudo codes of the HS and AS, the comments notation beside/after each if statement and while blocks indicate the maximum number of comparisons in each iteration, which are consumed by the CPU to execute these blocks. However, the AS shows 8 comparisons per iteration in the worst case and 5 comparisons per iteration in the best case, whereas the HS shows 4 comparisons per iteration in the worst case and 3 comparisons per iteration in the best case. Consequently, in the experimental performance test (Section 4.1.4), the HS shows
better performance for different key distributions when compared with the AS in the same test environment.

### 4.1.4 Results and experimental comparison

All search algorithms in this study are implemented in C++ using NetBeans 8.0.2 IDE based on the Cygwin compiler. The measurements are taken on a PC powered by a 2.1 GHz Intel machine with a Core i7 processor and 6 GB 1067 MHz DDR3 memory on the Windows platform. The experimental test was performed on empirical data that was generated randomly using a C++ library [79]. For all figures in this chapter, the execution times of the binary search, interpolation search, adaptive search and hybrid search were measured for 1000 repetitive runs of each algorithm on random arrays, whereas the size of the generated arrays ranged from 1000 to 23*106 elements. 1000 search keys were selected randomly from among the elements of the generated arrays; then these keys were searched using all the tested algorithms. This means that every algorithm was tested for successful searches only. The time readings were plotted after applying the smooth function in the MATLAB software, which uses a moving average filter in order to appear as smooth curve [83].

Measuring execution time is not a very accurate operation in a multitask/multithread operating system (OS) due to the execution time variations caused by uncontrollable hardware and software factors. Even we were running the same algorithm on the same data in the same environment [84,85]. Furthermore, in this test, we ran several algorithms on the multitask operating system, and there was no guarantee from the operating system to distribute CPU time slices among the tested algorithms fairly. We minimized this effect by configuring the operating system (Windows) on a fixed CPU frequency and set it to give the highest process priority for our testing program.

However, every tested algorithm was fast and consumed several milliseconds on the used hardware, so it was difficult to measure the execution time precisely in one execution. In order to reduce the tolerance error caused by execution time variations and to obtain more accurate time measurements, every time measurement was performed on repeated executions of the tested algorithms. The time measurement was
taken for 1000 times of running an algorithm for each array size with randomly selected search keys.

### 4.1.4.1 Performance Test on Uniform Distribution

In this test, we measured the execution time of all the tested algorithms on the uniformly distributed sorted array. In general, the interpolation search is the fastest algorithm for this distribution, and the binary search the slowest. However, it explains that HS performance is very close to the interpolation search at small sizes. Figure 4.2 shows that the HS was delayed only up to 6% behind the interpolation search for an array size of $23 \times 10^6$, while the AS was delayed by approximately 50% for the same array size.

![Figure 4.2](image.png)

**Figure 4.2** Execution-time of IntS, BS, AS and HS for small array size with uniform distribution.
4.1.4.2 Performance Test on Normal Distribution

In Figure 4.4, the interpolation search is obviously the slowest algorithm due to its complexity in non-uniform distribution data set. The HS shows better performance when compared with the AS in a normal distribution. The HS starts with an insignificant delay behind the BS. Gradually, the HS performance increases to 12% slower than the BS for array sizes less than 250,000 elements. However, the AS operates about 50% slower than the BS for the same size.

Figure 4.5 explains the performance of the HS for large array sizes up to $23 \times 10^6$ elements. Even though the HS operates more slowly than the BS, even up to 100% slower, the HS is still faster than the AS in this distribution.
Figure 4.4 Execution-time of BS, IntS, AS and HS for small array size with normal distribution.

Figure 4.5 Execution-time of BS, IntS, AS and HS for large array size with normal distribution.
The HS, similarly to the interpolation search, involves the data in a division (interpolation) process to estimate the location of the searched key. Thus, HS performance depends on the distribution of data and the length of the interpolated segment of data. On the other hand, the HS use a BS division method which does not rely on the data. As a consequence, this technique makes the behavior of the HS variant depend on the data length and data distribution. When the HS divides the given length into half, gradually the interpolation method becomes more efficient because the distribution of data has been changed during previous division processes. This explains why the HS showed better performance on small and moderate array sizes and poorer performance for large array sizes when compared with BS performance.

4.1.4.3 Performance Test for Exponential Distribution

The interpolation search shows even poorer performance when compared with its performance on the normally distributed array in the previous section. Therefore, we excluded the interpolation search from this test to enhance the readability of the graph.

In Figure 4.6, the HS displayed a very close performance comparison with the BS for array sizes less than 300,000 elements. Although the AS performed better when compared with its performance over the normally distributed array (see AS time values in Figure 4.5 and Figure 4.7), the AS was still slower than the BS by 35% and 75% for small and large array sizes, respectively (see AS performance in Figure 4.6 and Figure 4.7).

Figure 4.7 explains that the HS is the fastest algorithm for large array sizes in this distribution. On average, the HS is 10% faster than the BS for array sizes up to $23 \times 10^6$. 
Figure 4.6 Execution-time of BS, AS and HS for array small sizes with exponential distribution.

Figure 4.7 Execution-time of BS, AS and HS for large array sizes with exponential distribution.
4.2 Binary Search and Its Implementation Issue

In this work, the correct implementation of the binary search in [4] has been used. However, there is a common issue in the implementation of the binary search algorithm in the literature. This issue occurs when the first “if” statement is made to search for the desired key (containing the equality test only), and the second “if” statement is used to decide which half (right or left) will be selected for the next iteration; for instance, the weak implementation of the binary search in [28,86,87]. This drawback makes the binary search run the maximum number of comparisons per iteration. The analysis of the correct and weak implementation will be discussed in the analysis section. However, for the correct implementation, refer to the pseudo code in 2.2.1. Algorithm 9 is the pseudo code of weak implementation that is used in this study.

While the binary search was used as a search function in the Binary Search Tree (BST) data structure, we noticed the same issue occurs widely in the BST, as in the BST implementation in [27,88]. The author of [89] presented the true implementation of the BST in a recursive version and the weak implementation of the iterative version of the BST. This drawback also decreases the search speed in the binary search tree.
Algorithm 9: Weak Binary Search

1: procedure WBS( array, left, right, X)
2:     \textit{array} is the array that required to search
3:     \textit{left} is the index of left most element in array
4:     \textit{right} is the index of right most element in array
5:     \textit{X} is the element that we search for
6:     \texttt{Mid} $\leftarrow \frac{\texttt{left}+\texttt{right}}{2}$
7:     \textbf{while} \texttt{left} < \texttt{right} do
8:         if \textit{X} = \textit{array}[\texttt{Mid}] then
9:             \textbf{return} \texttt{Mid}
10:     \textbf{else if} \textit{X} < \textit{array}[\texttt{Mid}] then
11:         \texttt{right} $\leftarrow$ \texttt{Mid} − 1
12:     \textbf{else}
13:         \texttt{left} $\leftarrow$ \texttt{Mid} + 1
14:     \textbf{end if}
15:     \textbf{end while}
16:     \textbf{return} −1
17: \textbf{end procedure}

4.2.1 Binary Search Correct Implementation Analysis

We propose a precise method to analyze the binary search in terms of comparisons. This method thoroughly describes the behavior of the binary search. With the aid of this method, we can understand the complexity of the weak and correct implementations very well.

The author of [90] calculated the average number of iterations ($L$) for a successful search. He found $L = h − 1$ when the button level of the binary tree was full, and $L = h − 2h/n$ when the button level of the binary tree was not full. In both cases, $h = \log_2(n + 1)$, where $h$ is the height of the complete or perfect tree that represents the complexity of the binary search algorithm. By using this fact, we analyzed the binary search in term of the number of comparisons, as follows:
We assume that we are running the correct binary search algorithm in Section 2.2.1. Initially, each iteration consumes one comparison with a “while” statement. Then it may execute one or two comparisons in both “if” statements. If the first is true, the algorithm goes to the left child node in the tree (Figure 4.8) and consumes two comparisons for the current iteration in total. However, if the second condition is 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.

As explained in Figure 4.8, 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 is terminated at this node.

For the best case, the comparisons number of the correct binary search is equal to 3. However, we consider $h$ to be the height of the tree. The minimum number of comparisons at level $i$ is defined by:

$$C_{i,min} = 2i + 3 \quad \text{when} \quad 0 \leq i \leq h$$
However, the maximum number of comparisons at level $i$ is calculated as follows:

$$C_{\text{imax}} = 3i + 3 \quad \text{when } 0 \leq i \leq h$$

In case of the worst case, the correct binary search stops at the last level when $i = h = \log_2(n + 1)$. The maximum number of comparisons at this level is:

$$C_{\text{max}} = 3 \log_2(n + 1) + 3$$ \hspace{1cm} (4.1)

Eq. (4.1) computes the comparisons number of the correct binary search in the worst case for a successful search.

We assume every node at level $i$ has the same probability of access:

$$C_{i\text{avg}} = \frac{C_{\text{imin}} + C_{\text{imax}}}{2}$$

$$C_{i\text{avg}} = \frac{2i + 3 + 3i + 3}{2} = 2.5i + 3$$ \hspace{1cm} (4.2)

Eq. (4.2) calculates the average comparison number of the level ($i$). In the case of a successful search, the number of iterations may not reach the maximum level ($h$). As explained above, the average number of iterations which are calculated by [90] is equal to $L = h - 1$. Therefore, by substituting $L$ instead of $i$, we have:

$$C_{\text{avg}} = 2.5L + 3$$

$$C_{\text{avg}} = 2.5 \left( \log_2(n + 1) - 1 \right) + 3$$

$$C_{\text{avg}} = 2.5 \log_2(n + 1) + 0.5$$ \hspace{1cm} (4.3)

Eq. (4.3) calculates the average case comparisons number of the correct binary search.
4.2.2 Binary Search/Weak Implementation Analysis

Figure 4.9 shows the comparisons tree of the weak implementation of the binary search. The main reason for the weak implementation being 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, the branching to both children nodes consumes three comparisons (Figure 4.9).

For the best case, the weak binary search consumes only 2 comparisons. In Figure 4.9, we observe that the algorithm finds any element at level \( i \) consuming an equal number of comparisons, which is computed as follows:

\[
C_i = 3i + 2 \quad \text{when} \quad 0 \leq i \leq h
\]

For the worst case, the maximum number of comparisons occurs at the final level when \( i = h \log_2(n + 1) \).
\[ C_{\text{hmax}} = 3 \log_2(n + 1) + 2 \]  

Similarly, for the average case, \( L = h - 1 \), by substituting \( L \) instead of \( i \):

\[ C_{\text{avg}} = 3L + 2 \]

\[ C_{\text{avg}} = 3(\log_2(n + 1) - 1) + 2 \]

\[ C_{\text{avg}} = 3 \log_2(n + 1) - 0.5 \]  

Eq. (4.3) calculates the average case comparisons number of the weak binary search. However, by comparing Eq. (4.3) with Eq. (4.5), the binary search performance deteriorates with the mentioned weak implementation.

### 4.3 Traditional Ternary Search

The traditional ternary search is presented as an alternative to the binary search. Despite the traditional ternary search consuming a lower number of iterations compared to the binary search, the traditional ternary search has a higher number of comparisons per single iteration. This section explains the number of comparisons in detail.

In the literature, there are several studies presented for the ternary search, such as the analysis study in [91] and the pseudo-code which is presented in [92] as a ternary search. Additionally, there is a similar approach presented in [93].

Algorithm 10 is the traditional ternary search that used in this study.
Algorithm 10: Traditional Ternary Search

1: procedure TS( array, left, right, X)
2:     array is the array that required to search
3:     left is the index of left most element in array
4:     right is the index of right most element in array
5:     X is the element that we search for
6:     while left < right do
7:         Lci ← ⌈2(left+right)/3⌉
8:         Rci ← ⌈left+2×right/3⌉
9:         if X = array[Lci] then
10:            return Lci
11:         end if
12:         if X = array[Rci] then
13:            return Rci
14:         end if
15:         if X ≤ array[Lci] then
16:             right ← Lci
17:         else if X ≥ array[Rci] then
18:             left ← Rci
19:         else
20:             left ← Lci + 1
21:             right ← Rci−1
22:         end if
23:     end while
24:     return −1 ▷ not found
25: end procedure

The comparisons details are as follows:

1 comparison in the “while” statement in line #6

1 comparison in the “if” statement in line #9

1 comparison in the “if” statement in line #12

1 comparison in the “if” statement in line #15

1 comparison in the “if” statement in line #17
Total comparisons are 5 per iteration. Therefore, the maximum number of comparisons consumed by the traditional ternary search is $5 \log_3(n)$, while it is $3 \log_2(n)$ in the binary search. Consequently, the comparison number in the traditional ternary search is always higher than the comparison number in the binary search because $5 \log_3(n) > 3 \log_2(n)$.

The improved ternary search is designed to reduce the number of comparisons and to be faster than the correct binary search.

### 4.4 Improved Ternary Search

Algorithm 11 is the pseudo-code of 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 located at the Lci, the left third of the array will contain X. Correspondingly, if X is greater than the key located at Rci, the right third of the array will hold X. Otherwise, the middle third holds the required key X. These operations are repeated 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 the remaining keys to decide whether the search will finish successfully or unsuccessfully.
Algorithm 11: Improved Ternary Search

1: procedure ITS(array, left, right, X)
2:     array is the array that required to search
3:     left is the index of left most element in array
4:     right is the index of right most element in array
5:     X is the element that we search for
6:     while right - left > 2 do
7:         third $\left\lfloor \frac{\text{right} - \text{left}}{3} \right\rfloor$
8:         Lei $\leftarrow$ left + third
9:         Rci $\leftarrow$ right - third
10:        if X $\leq$ array[Lei] then
11:            right $\leftarrow$ Lei
12:        else if X $\geq$ array[Rci] then
13:            left $\leftarrow$ Rci
14:        else
15:            left $\leftarrow$ Lei + 1
16:            right $\leftarrow$ Rci - 1
17:        end if
18:     end while
19:     if X = array[left] then
20:         return left
21:     else if X = array[right] then
22:         return right
23:     else if X = array[left + 1] then
24:         return left + 1
25:     else
26:         return -1
27:     end if
28: end procedure

4.4.1 Improved Ternary Search Analysis

The improved ternary search reduces the average number of comparisons by reducing the cost of an iteration. 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.
If we assume \( j \) to be the position of the required element, \( C[j] \) is the number of comparisons required to retrieve the element at the \( j \) position. In each division process (iteration), there are only two possible states; if \( j \) is in the left third, the algorithm consumes 2 comparisons to go to the left part (1 comparison in “While” or base case, plus 1 comparison in the first “if” statement); therefore, \( C[j] = C[j] + 2 \). If \( X \) resides in the right third or middle third, the algorithm requires 3 comparisons to go to the corresponding part (previous comparisons plus 1 for the second “if” statement); therefore, \( C[j] = C[j] + 3 \).

The comparisons tree of the improved ternary search algorithm is shown in Figure 4.10. Walking to the left child node consumes two comparisons, whereas walking to the middle or right child node consumes three comparisons. However, the improved ternary search uses a linear search (in the last three “if” statements) to find \( X \) if \( n \leq 3 \) or the remaining number of elements is \( \leq 3 \).

![Figure 4.10 Improved ternary search comparisons tree](image)

However, by using a linear search on 3 elements, the comparison cost can be calculated as follows:
\[ C_{av3} = \frac{1 + 2 + 3}{3} = 2 \]

After the division process ends, the algorithm consumes 1 comparison to end the loop ("while" statement). Then, on average, it consumes 2 comparisons during the linear search \( C_{av3} \). Consequently, the algorithm adds 3 comparisons for the total number of comparisons that are consumed in the division process before it terminates.

If we assume \( n = 3^{k+1} \), \( k = \log_3(n) - 1 \). For Figure 4.10, the minimum number of comparisons at level \( i \) is:

\[ C_{imin} = 2i + 3 \quad \text{when} \quad 0 \leq i \leq k \]

The best-case comparison number of the improved ternary search at \( k \) is as follows:

\[ C_{min} = 2 \log_3(n) + 1 \]

while the maximum number of comparisons at level \( i \) is:

\[ C_{imax} = 3i + 3 \quad \text{when} \quad 0 \leq i \leq k \]

The worst-case comparison number of the improved ternary search at \( k \) is as follows:

\[ C_{max} = 3 \log_3(n) + 3 \]

We notice that the number of comparisons at each level is distributed non-uniformly. Therefore, it becomes difficult to calculate the exact average of comparisons at each level. However, we can calculate the approximate average number of comparisons at level \( i \) by carefully selecting four values as follows:

\[ C_{iavg} = \frac{C_{imin} + 1 + C_{imid} + C_{imax} - 1 + C_{imax}}{4} = \frac{C_{imin} + C_{imid} + 2C_{imax}}{4} \]

where

\[ C_{imid} = \frac{C_{imin} + 1 + C_{imax}}{2} = \frac{2i + 3 + 1 + 3i + 3}{2} = 2.5i + 3.5 \]
\[ C_{avg} = \frac{2i + 3 + 2.5i + 3.5 + 2(3i + 3)}{4} = 2.625i + 3.125 \]

at the last level \( i = k = \log_3(n) - 1 \)

\[ C_{avg} = 2.625 \log_3(n) + 0.5 \quad (4.6) \]

Since there is no possibility of the algorithm stopping before the final level, Eq. (4.6) is considered the average comparisons number for the improved ternary search.

Table 4.2 compares the complexity of the correct binary search and improved ternary search in terms of the comparisons number for the best, worst and average cases.

<table>
<thead>
<tr>
<th>Comparisons No.</th>
<th>Correct Binary Search</th>
<th>Improved Ternary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>3</td>
<td>( 2 \log_3(n) + 1 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( =1.82 \ln(n) + 1 )</td>
</tr>
<tr>
<td>Worst Case</td>
<td>( 3 \log_2(n+1) + 3 )</td>
<td>( 3 \log_3(n) + 1 )</td>
</tr>
<tr>
<td></td>
<td>( = 4.32 \ln(n+1) + 3 )</td>
<td>( =2.73 \ln(n)+1 )</td>
</tr>
<tr>
<td>Average Case</td>
<td>( 2.5 \log_2(n+1) + 0.5 )</td>
<td>( 2.625 \log_3(n) + 0.5 )</td>
</tr>
<tr>
<td></td>
<td>( = 3.6 \ln(n+1) + 0.5 )</td>
<td>( =2.39 \ln(n)+0.5 )</td>
</tr>
</tbody>
</table>

4.5 The Proposed Binary-Quaternary Search

The proposed algorithm is called the Binary-Quaternary (BQ) search, this algorithm exactly like the proposed ternary search regarding the implementation. The main difference being that the BQ search divides the length of the given array over four instead of three in the improved ternary search. Consequently, the behavior of the
algorithm completely changes. Figure 4.11 shows the behavior of the dividing technique in the BQ search.

When the required key $X$ is located in the left quarter ($x \leq array[Lci]$), the BQ search sets ($right = Lci$), which excludes $\frac{3}{4}$ of the length of the array for the next iteration. Similarly, for the right quarter, when $X$ is located in this quarter, the BQ search sets ($left = Rci$). In the case of $X$ residents in the middle half (between $Lci$ and $Rci$), the BQ search works like an ordinary binary search. However, the main benefit of BQ is that there is a $50\%$ probability of dividing the given length over $4$, thus consuming the same comparisons number in the binary search during each iteration. This approach reduces the iterations number remarkably compared with the binary search. In turn, it increases the performance of the BQ search. The experimental results explained in Section 4.7.2 show that the BQ search is faster than the binary search and faster than the improved ternary search for some conditions.

![Figure 4.11 BQ search dividing technique](image)

Algorithm 12 illustrates the pseudo code for the BQ search. The BQ initially calculates $Lci$, which indicates the end of the left quarter of the array and $Rci$, which indicates the beginning of the right quarter of the array.
4.5.1 BQ Search Analysis

The division technique of the BQ search randomly divides the given length in each iteration, either by 2 or 4. The BQ search continuously divides an array of N elements until the smallest size becomes less than or equal to 4. However, the worst case of the BQ occurs if all dividing operations are over 2. Hence, the BQ search needs $\log_2(n) - 2$
iterations to stop. Correspondingly, the best case occurs when all divisions are over 4. In this case, the BQ requires $\log_4(n) - 1$ iterations in order to end.

The calculation of the average case of the BQ search may be represented by the following mathematical problem. We assume N to be a positive number continuously divided over defined numbers. We then find the average of the consecutive division operations when each division operation divides the result of the previous operation randomly, either over 2 or 4 until achieving a result less than or equal to 4.

A probabilistic analysis is needed to compute the average case of the BQ because of the random division operations being over 2 or 4. Nevertheless, this section will walk through an example to understand the behavior of the algorithm; then it will determine an equation to compute the average case complexity of the BQ search precisely.

![Figure 4.12 BQ dividing tree](image-url)
Figure 4.12 shows the division tree of the BQ search for an array size $n = 256$, where branching to the left means dividing by 4 and branching to the right means dividing by 2.

For Figure 4.12, $n = 2^k$, $n = 256$, thus, $k = \log_2(n)=8$ where $k$ is the height of the tree. For any level ($i$), the number at a certain node can be obtained by dividing the number $n$ by the denominator of sub-divisions which will be either 2s or 4s until reaching this node. The BQ search stops when the value of the node reaches 64 or higher. That is because the remaining length of the array is less than or equal to four ($256/64 = 4$, $256/128 = 2$). Zeros in the tree mean the algorithm has stopped at a previous level. In other words, we will denote all the nodes at which the algorithm is stopped as a successful division.

The calculation of the average iterations number for Figure 4.12 requires finding the probability of a successful division at each level. However, we assume there is an array of $n$ elements, $k = \log_2(n)$. The average number of iterations is given by:

$$f(k) = \sum_{i=\lceil k/2 \rceil - 1}^{k-2} i \times P(i) \quad (4.7)$$

where $i$ is the level number, and $P(i)$ the probability of a successful division at level $i$.

For Eq. (4.7), the value of $i$ starts at $i = \lceil k/2 \rceil - 1$ because the probability of stopping at any level less than this value is zero. This means there is no possibility of running less than the best case. Similarly, $i$ should end at $k - 2$ (the worst case).

The most significant matter in the calculation of the level probability is finding the number of successful division operations at each level. A successful division operation is the division operation that terminates the BQ search. For our example, a successful division operation is the division operation (node) in each level that has a value equal to or higher than 64 (zero also means higher than 64).
Figure 4.13 shows the probability tree of the BQ when it is run on a 256-element array. \( X(i) \) is the number of successful division operations over a total number of divisions at level \( i \). The values of \( X(3) = 1/8 \), \( X(4) = 11/16 \) and so forth are obtained from Figure 4.12.

![Probability tree of BQ search](image)

The average number of iterations based on the probability tree in Figure 4.13 is computed as follows:

\[
\text{average} = 3 \, P(3) + 4 \, P(4) + 5 \, P(5) + 6 \, P(6)
\]

such that:

\[
P(3) = X(3) = \frac{1}{8} = 0.125
\]

\[
P(4) = X(4) \left(1 - X(3)\right) = \frac{11}{16} \times \frac{7}{8} = \frac{77}{128} = 0.6015625
\]

\[
P(5) = X(5) \left(1 - X(4)\right) \left(1 - X(3)\right) = \frac{31}{32} \times \frac{5}{16} \times \frac{7}{8} = \frac{1085}{4096} = 0.26481
\]
\[
P(6) = X(6) \left(1 - X(5)\right) \left(1 - X(4)\right) \left(1 - X(3)\right) = \frac{64}{64} \times \frac{1}{32} \times \frac{5}{16} \times \frac{7}{8} = \frac{2240}{262144}
\]
\[
= 0.0085
\]

To verify the result, \(P(3) + P(4) + P(5) + P(6)\) must be equal to 1.

\[
\text{average} = 3 \times 0.125 + 4 \times 0.6015625 + 5 \times 0.26481 + 6 \times 0.0085 = 4.156
\]

We perceived that the number of successful divisions at each level depended on the number of divisions over 4 in contrast with the number divisions over 2. The BQ search at an early stage reaches a successful division operation if the number of divisions over 4 is greater than the number of divisions over 2. However, the combinations of the number of divisions over 2 and 4 are invested to precisely calculate the number of successful divisions at each level. The following equation calculates \(X(i)\), which equals the number of successful divisions over the total number of divisions at level \(i\) for any \(k\):

\[
X(i) = \sum_{j=0}^{\left\lceil \frac{k}{2} \right\rceil} \frac{2^{(i-\left\lceil \frac{k}{2} \right\rceil-1)+\left\lfloor \frac{k}{2} \right\rfloor-\left\lfloor \frac{k}{2} \right\rfloor}}{2^i} \text{ where } \left(\left\lfloor \frac{k}{2} \right\rfloor - 1\right) \leq i \leq (k-2) \quad (4.8)
\]

From the probability tree in Figure 4.13, the probability of stopping at level \(i\) is defined by:

\[
P(i) = X(i) \times (1 - X(i-1)) \times (1 - X(i-2)) \times \ldots \times (1 - X\left(\left\lceil \frac{k}{2} \right\rceil - 1\right)) \quad (4.9)
\]

By substituting Eq. (4.8) into Eq. (4.9), we have:

\[
P(i) = \sum_{j=0}^{\left\lceil \frac{k}{2} \right\rceil} \frac{2^{(i-\left\lceil \frac{k}{2} \right\rceil-1)+\left\lfloor \frac{k}{2} \right\rfloor-\left\lfloor \frac{k}{2} \right\rfloor}}{2^i} \prod_{m=\left\lfloor \frac{k}{2} \right\rceil-1}^{m=i-1} \left(1 - \sum_{p=0}^{\left\lceil \frac{k}{2} \right\rceil} \frac{\left(\left\lceil \frac{k}{2} \right\rceil - 1\right)^p}{2^m}\right) \quad (4.10)
\]
To calculate the average number of iterations that are consumed by the BQ search, we substitute (4.10) into (4.7), as follows:

\[
\begin{align*}
f(k) &= \sum_{i=[k/2]-1}^{k-2} i \ast \sum_{j=0}^{[k/2]} \frac{(i)}{2^j} \\
&\ast \prod_{m=[k/2]-1}^{m=i-1} \left(1 - \sum_{n=0}^{2([m/2] - 1)+[k/2]-[k/2]} \left(\frac{m}{2^m}\right)\right)
\end{align*}
\]  

(4.11)

The average case of the BQ (in terms of iterations) for many values of \( k \) is calculated in Figure 4.14 by using Eq. (4.11). The figure shows that the BQ needs a number of iterations closer to \( \log_3(n) \) than \( \log_2(n) \), which means that the BQ search average case complexity is closer to its best case than its worst case. In contrast, the binary search average case is closer to its worst case of \( \log_2(n) \).

![Figure 4.14 BQ average case iterations](image-url)
Despite the BQ search showing slightly fewer average iterations compared with the improved ternary search, the BQ search unfortunately consumes a slightly higher average comparison number than the improved ternary search. The average number of comparisons of the BQ search may be obtained as follows:

\[ BQ \text{ average comparisons number} = \text{average comparisons of dividing process} + \text{average of linear search}. \]

The average cost of the linear search to search for 4 elements may be obtained similarly to improved the ternary search, where:

\[ C_{av4} = \frac{1 + 2 + 3 + 4}{4} = 2.5 \]

The BQ search consumes 1 comparison to terminate the dividing process. Therefore, the total average comparisons number of the linear search will be 3.5 comparisons.

For each iteration, the dividing process consumes 2 comparisons within a probability of occurrence of 25\% for the left quarter and 3 comparisons within the probability of occurrence of 75\% for the middle half and the right quarter. The BQ search consumes \( f(k) \) iterations on average; therefore, we have the following:

\[ BQ \text{ average comparisons number} = (2 \ast 0.25 + 3 \ast 0.75) f(k) + 3.5 \]

\[ BQ \text{ average comparisons number} = 2.75 f(k) + 3.5 \quad (4.12) \]

Figure 4.15 presents the average comparisons number of the binary search, improved ternary search, and BQ search. Where Eq. (4.12) used in the graph as BQ average comparisons number.

In conclusion, the BQ search remarkably consumes a lower number of iterations and comparisons compared to the binary search. On the other hand, despite BQ search theoretically consumes average comparisons number slightly greater than the improved ternary search, experimental results showed that the BQ search is faster than
the improved ternary search in some conditions. The next section will explain the cause of the differences in the performances experimentally.

Figure 4.15 Average comparisons numbers of the binary, improved ternary and BQ searches

4.6 Implementation of Improved Ternary Search vs. BQ Search

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 “Third = (right-left)/4;” in the BQ search were compiled into assembly language by the compiler, as in Table 4.3. 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 BQ search. Moreover, the assembly code in the BQ search was smaller than that in the improved ternary search. In another word, division over 4 is faster than division over 3. In turn, this increases the performance of the BQ search compared to the improved ternary search. In brief, the BQ search showed better performance compared to the improved ternary search if the cost of a comparison operation is less than the cost a division operation.

Table 4.3 Compilation of C++ instruction into assembly language.

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

The experimental environment of this test is the same software and hardware configuration as those that were used in Section 4.1.4. However, two types of generated data are used: a numeric array of 8-byte numbers (double) and a text array of 50 characters’ key length.

4.7.1 Execution Time of Weak and Correct Binary Searches

Experimental results showed that the difference between the weak and correct implementations was not detected in our test environment when the 8-byte key was used. Figure 4.16 explains the experimental execution time for the weak and correct implementations of the binary search for the 50-byte key length. The figure shows that there is a small gain in execution time for the small size array and the gain increases when the array size increases.

![Figure 4.16 Binary search weak/correct implementation execution time](image_url)
4.7.2 Execution Time of Binary, Improved Ternary and BQ Searches

The cost of a comparison process obviously affects the performance of the algorithms being tested. This cost is influenced by data type and hardware considerations. For instance, the computer needs more time to compare two strings of 50 bytes than two numbers of 8 bytes. Furthermore, the cost of any comparison increases if the time to access the main memory increases. The other case that has a higher cost for the comparison process is the external search. An external search is a search when the array size is greater than the main physical memory or available memory. Additionally, the access to secondary storage devices increases the time of the comparison process.

Our proposed algorithms (Ternary and BQ searches) both have fewer comparisons relative to the binary search. However, their drawback is that they have more computation variables relative to the binary search. On the other hand, their advantages include the fact that the cost of calculating these variables does not depend on the data type or internal/external memory access operations. Another advantage is the limited number of variables involved in this computation, so cache memories or CPU registers can hold these variables to reduce the access time to these variables. because of the frequent access to these variables.
Figure 4.17 shows the experimental execution time of the Binary, Ternary and BQ searches for the 8-byte double data type. We see that the ternary search execution time is equal to or less than the time consumed by the binary search for moderate sized arrays. The gain in time increases when the size of the searched array increases. However, the BQ search delivers better performance than the improved ternary search and the binary search for all array sizes.
CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, several solutions related to the sorting and searching problem have been presented. We started by presenting the new Bidirectional Conditional Insertion Sort (BCIS) algorithm. The performance of the proposed algorithm has significant improvements over the standard insertion sort. Results showed that BCIS has an average case of approximately $O(n^{1.5})$. Additionally, BCIS maintains the fast performance of the best case of a classical insertion sort when running on the already sorted array, since BCIS time complexity is $(4n)$ over that array. Moreover, the worst case of BCIS is better than the insertion sort while BCIS consumes only $n^2/6$ comparisons with the reverse sorted array.

The other advantage of BCIS is that the algorithm is faster than QuickSort for relatively small arrays (up to 1500). This feature does not only make BCIS the best solution for relatively small arrays. It makes BCIS a powerful interesting algorithm to use in conjunction with QuickSort.

In addition, four search solutions were presented in this thesis. First, we proposed a hybrid search algorithm called HS. The proposed algorithm was designed to be used on the unknown distributed sorted array. This algorithm is based on the principle of the binary and linear interpolation search. The proposed algorithm was compared with three algorithms used for a search on an ordered list. The experimental results explained that HS showed a performance close to the fastest algorithm in each tested distribution, especially when HS was run on small or moderate size arrays. Hence, we recommend using HS as a general search algorithm for unknown distributed ordered lists within a given situation that shows acceptable performance.

Second, we analyzed the binary search algorithm in terms of comparisons followed by addressing a common issue in the implementation of a binary search, which reduced
the performance. The binary search implementation of low performance is called weak implementation. The analysis proved that the weak implementation consumes the same number of comparisons in both the average and worst cases. Meanwhile, in the correct implementation, the number of comparisons in the average case is less than that number in the worst case. In other words, the weak implementation always makes the binary search run in the worst case. We studied the occurrence of the same implementation issue in other algorithms or programs.

Third, despite the presence of a traditional ternary search implementation that consumes a lower iterations number compared with the binary search, this implementation is still slower than the binary search because of the high cost per iteration. We presented a new efficient implementation of the ternary search algorithm. The improved ternary search has been analyzed and compared theoretically and experimentally with the binary search. Comparison results showed that the proposed algorithm is faster than the binary search. However, our improvement on the ternary search was obtained by reducing the number of comparisons per iteration.

Finally, we proposed a new Binary-Quaternary (BQ) search. The proposed algorithm is used to search ordered lists. The BQ search 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 a BQ search has lower average comparison numbers than the binary search and slightly higher numbers than the improved ternary search. However, our experimental results showed that the BQ search is faster than the improved ternary search when the cost of a comparison operation is lower than the cost of a division operation.

5.2 Future Work

The research presented in this thesis seems to have raised more questions than it has answered. There are several lines of research arising from this work which should be pursued.

We recommend a redesign of all presented sorting and searching algorithms to invest in the hybrid hardware architecture recently produced by Intel (mentioned in
Section 1.2.2). Since sorting and searching algorithms are frequently used in practice, the embedded FPGA accelerator within Xeon processors was designed to increase the performance of frequently used programs.

Additionally, for the BCIS sorting algorithm presented in Chapter Three, we believe that it is easy to produce a parallel BCIS because the BCIS can insert items into the left and right part of an array simultaneously. However, a parallel BCIS can be tested on RISC or CISC computers. A parallel BCIS is expected to show better performance with multi-core CPUs compared with single-core CPUs.

Finally, for the search algorithms presented in Chapter 4, namely hybrid search, correct and weak binary search, improved ternary search and BQ search, we recommend performing an experimental comparative study of these algorithms when they work in external searches or in a distributed data environment.
REFERENCES


[41] Grabowski, F. & Strzalka, D., Dynamic behavior of simple insertion sort


[85] Mazouz, A. & Barthou, D., Performance Evaluation and Analysis of Thread


CURRICULUM VITAE

PERSONAL INFORMATION

Name Surname : Adnan ALAJEELI

Date of Birth: 16/10/1977
Phone: +90 553 363 77 24
E-mail: adnshr@gmail.com

EDUCATION

High School: Ibn Haldon High School- Kirkuk-Iraq
Bachelor : (1999) Northern Technical University, Engineering Technical College of Mosul. Mosul-Iraq
Master Degree : Universiti Tenaga Nasional, College of Graduate Studies- Malaysia

WORK EXPERIENCE

September 2007 - Engineering Dep. - Salah Eldeen governorate - Iraq
Present
April 2000 – Office of Engineering Affairs, Baghdad- Iraq
August-2007

TOPICS OF INTEREST

-Computer Algorithm
- Computer Network
- Data Structure
- Artificial intelligence