An Advanced Research on Enhancing Sorting and Searching Algorithms: A Comprehensive Study

[1] Bishal Sarkar, [2] Balaka Sarkar, [3] Arpita Bhattacharjee, [4] Banisha Adhikary, [5] Biswambhar Saha, [6] Surajit Basak and [7] Kaushik Roy

Department of Electronics and Communication Engineering, Guru Nanak Institute of Technology, Kolkata, India

[1]sarkarbishal93300@gmail.com, [2]balaka1406@gmail.com, [3]arpitabhattacharjeeeducation@gmail.com, [4]banishaadhikary@gmail.com, [5]biswambharsaha08@gmail.com, [6]surajit.basak@gnit.ac.in and [7]kaushik.roy@gnit.ac.in

***Abstract*—This research paper delves into the fundamental concepts of searching and sorting algorithms, pivotal components in computer science and information technology. The abstract begins with exploring the significance and ubiquity of these algorithms in various computing applications. A comprehensive examination elucidates the essential properties inherent in both searching and sorting algorithms, elucidating their distinct characteristics and functionalities. The paper provides a detailed portrayal of the working mechanisms of different searching and sorting algorithms, elucidated through pictorial representations for enhanced comprehension. Each algorithm is meticulously dissected, offering clear insights into its operational logic and efficiency. Additionally, the inclusion of algorithmic pseudocode and C code implementations enables practical understanding and facilitates implementation in real-world scenarios. A pivotal aspect of this research is the comparative analysis of the time and space complexities associated with each algorithm. Through meticulous evaluation, the paper elucidates the performance disparities among various algorithms, aiding researchers and practitioners in making informed decisions regarding algorithm selection based on specific application requirements. Furthermore, the paper underscores the practical implications of these algorithms in modern computing paradigms, emphasizing their indispensable role in optimizing computational tasks across diverse domains. By providing a holistic overview of searching and sorting algorithms, this research paper serves as a valuable resource for both novices and seasoned professionals seeking to deepen their understanding and proficiency in algorithmic design and analysis.**

***Keywords*— Data Structures, Searching Algorithms, Sorting Algorithms, Complexity Analysis, Binary Search, Linear Search, Quick Sort, Merge Sort, Insertion Sort, Selection Sort, Bubble Sort.**

I. INTRODUCTION

In the ever-expanding landscape of computational sciences, the efficiency of algorithms plays a pivotal role in shaping the performance of various applications. Among the foundational elements are searching and sorting algorithms, essential tools that contribute to the optimization of data retrieval and organization. This research delves into the comprehensive analysis of diverse searching and sorting algorithms, exploring their properties, working mechanisms, and the intricacies of their implementation in the C programming language.

As the backbone of many computing tasks, searching algorithms determine the effectiveness of locating specific elements within datasets, while sorting algorithms establish the order and arrangement of information for streamlined processing.

Moreover, this research extends beyond algorithmic descriptions, delving into the practical implications of each method by presenting the corresponding C code. The code snippets offer a tangible understanding of the algorithms' implementation, fostering clarity and accessibility for researchers and practitioners alike.

A crucial facet of this exploration involves a comparative analysis of time and space complexities. By systematically evaluating these metrics, the research aims to provide a nuanced understanding of the trade-offs inherent in each algorithm, facilitating informed decision-making in algorithm selection for specific applications.

A. Motivation:

The motivation for this research paper on searching and sorting algorithms stems from the pressing need to address the challenges posed by the escalating volume of data in the field of computer science. Observing the direct impact of algorithmic efficiency on system performance in practical scenarios fuelled my interest in delving into the intricacies of these fundamental algorithms. The aim is to contribute practical insights that can guide informed decisions in optimizing computational tasks and applications. This research paper is committed to advancing the understanding of algorithmic implications in real-world computing challenges.

B. Objectives:

1. Examine different sorting algorithms, such as bubble sort, insertion sort, merge sort, and quicksort, to understand their mechanisms and assess their efficiency in terms of time and space complexity.
2. Provide a comprehensive pictorial representation of the working process for each algorithm, enhancing the visual understanding of their functionality.
3. Present detailed algorithmic descriptions and C code implementations for each searching and sorting algorithm, facilitating practical application and implementation by researchers and practitioners.

4. Conduct a comparative analysis of the time and space complexity of the investigated algorithms, aiming to identify trade-offs and optimal use cases for each algorithm in practical scenarios.

C. *Contribution:*

Our research provides a deep insight into the mechanisms of various commonly used sorting and searching algorithms which are explained using simple diagrams. Visual aids enhance understanding by providing insights into the internal steps and decision-making processes of each algorithm. In contrast, many existing papers [2] on this subject often lack such visual representations, relying solely on textual descriptions that may be challenging for readers to conceptualize. One notable aspect of our paper is the detailed examination of the properties of each algorithm and a comparison of their time and space complexity, providing readers with practical insights into the strengths and weaknesses of various approaches. This gives our paper a preference over others like [16] as they lack. Moreover, our provision of algorithmic implementations accompanied by C code offers practical utility, allowing readers to directly apply the discussed algorithms in their projects and enhancing its value as a practical resource for researchers, students, and practitioners in the field. This hands-on approach distinguishes our paper from others like [5] as they are deficient in such resources.

II. SORTING

Sorting algorithms are crucial components in computer science, used to arrange elements in a specific order within a dataset. Sorting allows for the systematic arrangement of data in a specific order, making it easier to search, retrieve, and manage information efficiently. Once data is sorted, search operations become more efficient [15]. Binary search, for instance, relies on sorted data, enabling faster retrieval of information compared to unsorted datasets. Sorted data is often presented in a more visually appealing and comprehensible manner. Sorting is fundamental in database management systems enabling the efficient execution of queries, indexing, and joins, contributing to the overall performance of database operations.
Types of Sorting: Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Quick Sort [11].

A. *Selection sort*

Selection sort is one of the easiest sorting techniques. The algorithm functions by partitioning the provided list of elements into two segments: a sorted section and an unsorted section. Initially, the entire list comprises the unsorted segment, while the sorted portion remains empty. During each iteration, it identifies the minimum (or maximum) value from the unsorted section and swaps it with the first element of the unsorted portion, thereby incorporating it into the sorted list. If there are n elements in the given list of elements then the algorithm will require n-1 passes to sort the elements. The visual representation of the working of Selection sort is shown in Fig.1.

- *Properties:*
  1. In the worst case, it performs n-1 swaps for an array having n unsorted elements.
  2.It makes n(n-1)/2 comparisons for an array having n unsorted elements in the worst case.
  3. Selection sort is an In-place sorting algorithm so it does not take additional memory space and sort the elements withinthe given array without using extra memory.
  4. Selection sort is not an adaptive algorithm as its performance remains unaffected by the initial order of the input elements.
  5. It is not a stable algorithm as it may change the relative order of equal elements.

Fig.1 Pictorial representation of Selection Sort [4].

- *Algorithm:*
  Step 1: Scan the elements from left to right in the unsorted part to find the minimum element, assuming the current element to be minimum.
  Step 2: If a smaller element is found in the unsorted part, update the minimum.
  Step 3: Swap the minimum element found in the unsorted part with the first element of the unsorted part.

Step 4: The first element of the unsorted array is now considered as a part of the sorted subarray and the unsorted part will now be the entire array except the sorted subarray.
Step 5: Repeat steps 1 to 4 for the remaining unsorted part of the array and continues till the unsorted part is left with one element or the entire array is sorted.

- *Program:* Here is an implementation of the Selection sort algorithm in C language.

```
#include <stdio.h>
int main()
{
   int n,i,j,min,t;
printf("Enter the number of elements :  ");
   scanf("%d",&n);
   int A[n];
printf("\nEnter the elements of the array : \n");
   for(i=0;i<n;i++)
 scanf("%d",&A[i]);
   for(i=0;i<n-1;i++)
    {
      min=i;
      for(j=i+1;j<n;j++)
      {
         if(A[j]<A[min])
         min=j;
      }
      t=A[i];
A[i]=A[min];
      A[min]=t;
   }
printf("\nThe sorted array is : \n");
for(i=0;i<n;i++)
printf("%d  ",A[i]);
return 0;
}
```

B. Bubble sort

Bubble sort is considered as the simplest or easiest sorting technique. The basic working principle of this sorting technique is that it checks if its adjacent elements are already arranged in the assigned order and if it matches it leads to the next iteration else it swaps the elements. Bubble sort can be optimized [17] by including a mechanism to check whether any swaps were made during a pass, if no swaps are detected in a pass, then the algorithm terminates early giving us the sorted list. This modification makes it a bit more efficient for partially sorted lists by eliminating unnecessary iterations. The visual representation of the working of Selection sort is shown in Fig.2.

- *Properties:*
  1. In the worst case, it performs approximately $n^2/2$ swaps for a list of n elements.
  2. It makes $n^2/2$ comparisons in the worst-case and n-1 comparisons in the best-case scenario.
  3. Bubble sort is an in-place sorting algorithm so it doesn't require additional memory for sorting.
  4. Bubble sort is adaptive as its performance can be improved in partially sorted lists.
  5. It is a stable sorting algorithm so equal elements maintain their relative order.

Fig.2 Pictorial representation of Bubble Sort [6].

- *Algorithm:*
  Step 1: Begin with the first element of the list [18].
  Step 2: Compare the current element with the next element. If the current element is greater than the next one, swap them.
  Step 3: Move to the next pair of elements and repeat step 2 until the end of the list is reached.
  Step 4: After completing one pass through the list, the largest element is now at the end of the list.
  Step 5: Repeat steps 1-4 for the remaining unsorted elements until the entire list is sorted [6].

- *Program:* Here is an implementation of the Bubble sort algorithm in C language.

```
#include <stdio.h>
int main() {
 int n,i,j,t;
printf("Enter the number of elements : ");
 scanf("%d",&n);
int  A[n];
printf("\nEnter the elements of the array : \n");
for(i=0;i<n;i++)
scanf("%d",&A[i]);
    for(i=0;i<n-1;i++)
{
       for(j=0;j<n-1-i;j++)
{
          if(A[j]>A[j+1])
{
              t=A[j];
              A[j]=A[j+1];
              A[j+1]=t;
}
}
  }
printf("\nThe sorted array is : \n");
 for(i=0;i<n;i++)
printf("%d  ",A[i]);
return 0;
 }
```

C. Insertion sort

Insertion sort is a simple sorting algorithm that builds the sorted array one element at a time. It divides the list of elements into a sorted part and an unsorted part such that the first element of the list is a sorted part and the rest is the unsorted part. It is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e., the position to which it belongs in a sorted array. It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element in the sorted array to one position ahead. The visual representation of the working of Selection sort is shown in Fig.3.

- *Properties:*
  1. In the worst case, Insertion sort makes $n^2/2$ swaps for a list of n elements.
  2. It makes $n^2/2$ comparisons in the worst-case and n-1 comparisons in the best-case scenario.
  3. Insertion sort is an in-place sorting algorithm so it doesn't require additional memory for sorting.
  4. Insertion sort is adaptive as its performance can be improved in partially sorted lists.
  5. It is a stable sorting algorithm so equal elements maintain their relative order.

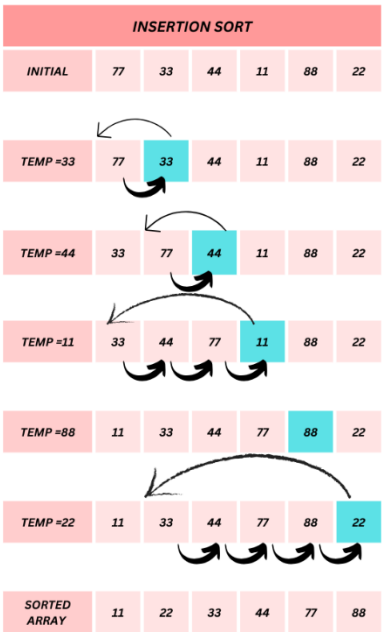

Fig.3 Pictorial representation of Insertion Sort [6].

- *Algorithm:*
  Step 1: Begin with the second element (index 1) assuming that the first element is already sorted [18].
  Step 2: Compare the current element with the elements before it.
  Step 3: Move the current element backward until finding the correct position where it is greater than the element before it and smaller than the element after it.
  Step 4: Insert the current element into its correct position.
  Step 5: Repeat steps 2-5 for the remaining elements of the unsorted part.
  Step 6: After iterating through all elements, the array is sorted [6].

- *Program:* Here is an implementation of the Insertion sort algorithm in C language.

```
#include <stdio.h>
int main()
{
  int n,i,ptr,temp;
  int arr[50];
printf("Enter number of elements : ");
  scanf("%d",&n);
printf("Enter the elements of array: \n");
  for(i=0;i<n;i++)
     scanf("%d",&arr[i]);
  for(i=1;i<n;i++)
 {
      temp=arr[i];
ptr=i-1;
 while(ptr>=0 &&arr[ptr]>temp)
{
arr[ptr+1]=arr[ptr];
ptr--;
     }
arr[ptr+1]=temp;
   }
printf("Sorted list in ascending order : \n");
  for(i=0;i<=n-1;i++)
printf("%d ",arr[i]);
  return 0;
}
```

D. Merge sort

Merge sort is a popular recursive algorithm that follows the divide-and-conquer paradigm. It works by dividing the input array into two halves, sorting each half recursively, and then merging the sorted halves to produce a single sorted array. Thus, the key operation in this sorting technique is merging two sorted arrays into a single sorted array. This process involves comparing elements from the two arrays and arranging them in the correct order. The visual representation of the working of Selection sort is shown in Fig.4.

- *Properties:*
  1. In Merge sort, the concept of "swaps" is not typically used, as the primary operation is merging rather than swapping elements directly.
  2. For any list of n elements, Merge sort has a total of nlogn comparisons [14].
  3. It is not an in-place sorting technique as it requires additional memory for the temporary arrays used during the merging step.
  4. Merge sort is adaptive and performs consistently even for a partially sorted list of elements.
  5. It is a stable sorting algorithm as the relative order of equal elements is maintained after sorting.

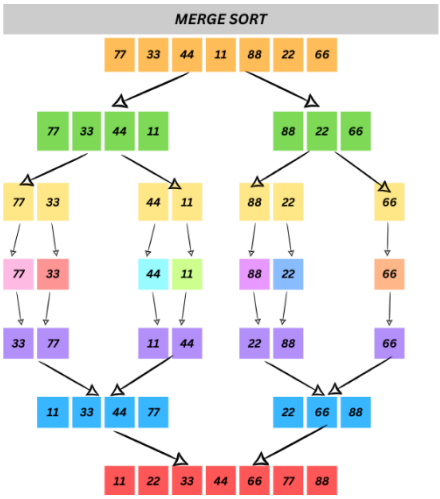

Fig.4 Pictorial representation of Merge Sort [4][8].

- *Algorithm:*
  Step 1: Base case:If the given list of elements has only one element or is empty, it is already sorted, and no sorting is required.
  Step 2: Divide:Divide the unsorted list into two halves by finding the midpoint.
  Step 3:Recursion: Apply the merge sort algorithm recursively to each half of the divided list. This involves repeating the steps for the left and right halves until the base case is reached.
  Step 4: Recursion: Apply the merge sort algorithm recursively to each half of the divided list. This involves repeating the steps for the left and right halves until the base case is reached.
  Step 5: Repeat:Repeat steps 2-4 recursively until the entire list is sorted.

- *Program:* Here is an implementation of the Merge Sort algorithm in C language [8].

```
 #include <stdio.h>
  void print(int *A,int n)
  {
for(int i=0;i<n;i++)
  printf("%d  ",A[i]);
printf("\n");
      }
      void merge(int A[],int lb,intmid,intub)
  {
     int i,j,k;
     int B[100];
  i=lb;
  j=mid+1;
  k=lb;
  while(i<=mid && j<=ub)
     {
     if(A[i]<A[j])
      {
         B[k]=A[i];
```

```
        k++;
i++;
     }
     else
     {
        B[k]=A[j];
        k++;
        j++;
     }
   }
   while(i<=mid)
   {
        B[k]=A[i];
        k++;
i++;
   }
   while(j<=ub)
   {
        B[k]=A[j];
        k++;
        j++;
   }
for(int i=lb;i<=ub;i++)
      A[i]=B[i];
}
void mergesort(int A[], int l, int h)
{
   if (l < h)
   {
      int m = (l + h)/2;
mergesort(A, l, m);
mergesort(A, m + 1, h);
merge(A, l, m, h);
   }
}
int main()
{
   int n,low,high;
printf("Enter the size of array : \n");
   scanf("%d",&n);
   int A[n];
printf("Enter the elements in the array : \n");
for(int i=0;i<n;i++)
      scanf("%d",&A[i]);
   low=0;
   high=n-1;
mergesort(A,low,high);
printf("The sorted array is : \n");
   print(A,n);
   return 0;
}
```

E. Quicksort

Quick sort is a commonly used sorting algorithm based on the *Divide & Conquer* approach and is often opted over other sorting algorithms based on its competence and effectiveness. It proceeds by choosing a pivot element from an array and divides it into two subarrays – one with elements smaller than the selected pivot and the other with elements bigger than it. The algorithm then recursively sorts the subarrays until the complete array is sorted. The visual representation of the working of Selection sort is shown in Fig.5.

- *Properties:*
  1. Quick sort rearranges elements by partitioning the array around a pivot element so the number of swaps in this sort is not explicitly counted as the primary operation is partitioning, which may involve swapping elements to ensure smaller elements on the left and larger elements on the right of the pivot.

2. It makes $O(n^2)$ comparisons in the worst case (where pivot element selection leads to unbalanced partitions) and O(nlogn) comparisons in the best case, for a list of n elements.
3. Quick sort is an in-place sorting algorithm so it doesn't require additional memory for sorting.
4. Quick sort is not a stable algorithm as it may change the relative order of equal elements during rearrangements in partitioning.

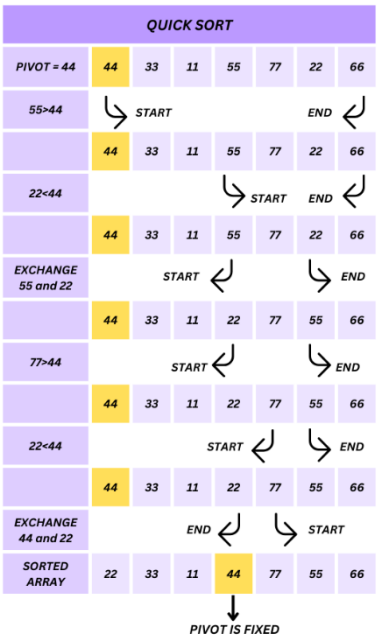

Fig.5 Pictorial representation of Quick Sort [4].

- *Algorithm:*
  Step 1: Select a pivot element from the array. The choice of pivot can affect the algorithm's performance.
  Step 2: Rearrange the array elements such that elements smaller than the pivot are on the left, and elements greater than the pivot are on the right [14].
  Step 3: The pivot is now in its final sorted position.
  Step 4: Repeat steps 1-4 to the subarrays on the left and right of the pivot such that the subarray has at least 2 elements.
  Step 5: After all recursive calls are complete, the entire array is sorted [9].

- *Program:* Here is an implementation of the Quick Sort algorithm in C language.

```
#include <stdio.h>
void swap(int *a,int *b)
{
int temp;
temp=*a;
*a=*b;
*b=temp;
 }
int partition(int C[],int l,int u)
{
   int pivot,start,end;
   pivot=C[l];
   start=l;
   end=u;
   while(start<end)
{
      while(C[start]<=pivot)
         start++;
     while(C[end]>pivot)
         end--;
  if(start<end)
 swap(&C[start],&C[end]);
 }
      swap(&C[l],&C[end]);
      return end;
```

```
}
  void quick(int B[],int lb,int ub)
{
  if(lb<ub)
{
 int loc=partition(B,lb,ub);
     quick(B,lb,loc-1);
  quick(B,loc+1,ub);
}
}
   int main()
{
  int n,low,high;
printf("Enter the size of array : \n");
     scanf("%d",&n);
     int A[n];
printf("Enter the elements in the array : \n");
for(int i=0;i<n;i++)
scanf("%d",&A[i]);
     low=0;
 high=n-1;
     quick(A,low,high);
printf("The sorted array is : \n");
for(int i=0;i<n;i++)
printf("%d ",A[i]);
   return 0;
}
```

F. Complexity in sorting algorithms

The time and space complexity of the sorting algorithms is commonly represented using Big O notation, which describes the upper bounds of an algorithm's growth rate as a function of n, which is the number of elements in the given list, as shown in TABLE I.

TABLE I : Time and space complexity of Sorting algorithms [10].

ALGORITHMS	**TIME COMPLEXITY**			**SPACE COMPLEXITY**		
	Best Case	**Average Case**	**Worst Case**	**Best Case**	**Average Case**	**Worst Case**
SELECTION SORT	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	O(1)	O(1)
BUBBLE SORT [18]	O(n)	$O(n^2)$	$O(n^2)$	O(1)	O(1)	O(1)
INSERTION SORT [18]	O(n)	$O(n^2)$	$O(n^2)$	O(1)	O(1)	O(1)
QUICK SORT [1][14]	O(nlogn)	O(nlogn)	$O(n^2)$	O(logn)	O(logn)	O(n)
MERGE SORT [1][14]	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	O(n)	O(n)

III. SEARCHING

Searching algorithms play a crucial role in computer science and information retrieval systems, as they are designed to efficiently locate specific items within datasets.These algorithms play a pivotal role in various applications, from databases to web search engines. Searching is often a fundamental step in sorting algorithms and ranking systems, enabling the identification of the position or existence of elements within an ordered set [15]. Many computational problems involve searching for a solution within a given space of possibilities. Searching algorithms play a critical role in solving these problems efficiently [3]. Types of searching: Linear Search, Binary Search.

A. Linear Search

Linear search, also known as sequential search, is a basic algorithm employed to locate a specific element within a list or array [12]. The process involves scanning the elements one by one, starting from the beginning, until a match is found or the entire list has been traversed. This search method applies to both sorted and unsorted lists, although it is most practical for unsorted data due to its simplicity [13]. The visual representation of the working of Selection sort is shown in Fig.6.

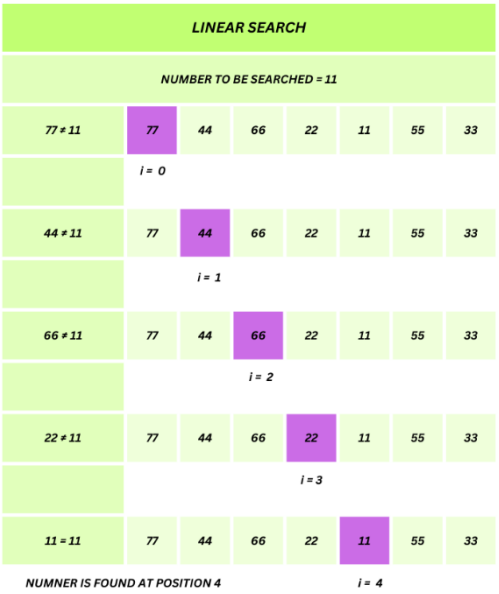

Fig.6 Pictorial representation of Linear Search.

- *Algorithm:*
  Step 1: Begin at the first element of the list.
  Step 2: Check if the current element matches the target element.
  Step 3: If there is a match, the search is successful, and returns the index of the element [2].
  Step 4: If there is no match and there are more elements in the list, move on to the next element.
  Step 5: Repeat steps 2-4 till the end of the list is reached.
  Step 6: If the entire list has been traversed and no match is found, it indicates that the target element is not present in the list.

- *Program:* Here is an implementation of the Linear search algorithm in C language.

```
#include <stdio.h>
int main()
{
    int i,k=0,ns,n;
    printf("Enter the number of elements\n") ;
    scanf("%d", &n) ;
      int a[n];
    printf("Enter %d elements:\n", n) ;
    for(i=0;i<n;i++)
     scanf("%d",&a[i]);
    printf("Enter the number to be searched :\n");
      scanf("%d",&ns);
    for(i=0;i<n;i++)
     {
    if(a[i]==ns)
    {
    k=1;
    break;
      }
     }
    if(k==1)
    printf("The number is present at position %d.",i);
    else
```

```
        printf("The number is not present.");
        return 0;
}
```

B. Binary Search

Binary search is a clever algorithm used to find a specific element in a sorted list of elements. Instead of searching through each element one by one, binary search splits the list In half and checks if the element we're looking for is in the first or second half. It keeps dividing the list until it finds the element or determines it's not there [12]. It's like finding an element in a dictionary by flipping to the middle page and then narrowing it down. The time complexity of binary search is logarithmic, specifically O(log n) [13], where n is the number of elements in the sorted array. This means that the search time grows slowly even as the size of the array increases. It's one of the reasons why binary search is so efficient [3].The visual representation of the working of Selection sort is shown in Fig.7.

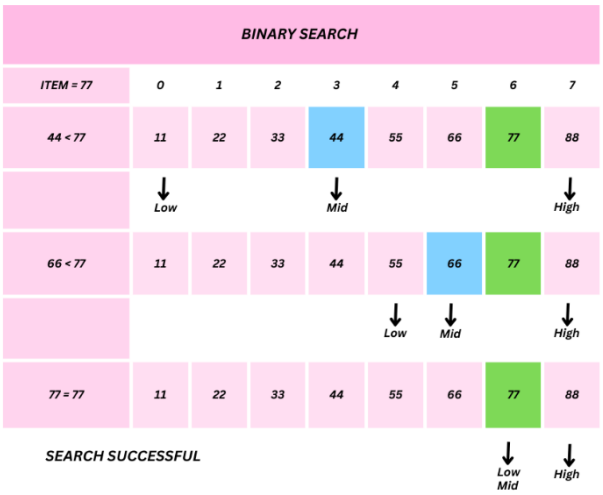

Fig.7 Pictorial representation of Binary Search.

- *Algorithm:*
  Step 1: Start with a sorted array.
  Step 2: Set two pointers, low and high, to the beginning and end of the array, respectively.
  Step3: Calculate the middle index as (low + high) / 2.
  Step 4: Compare the middle element with the target value.
  Step 5: If they are equal, the target value is found so return the index [2][7].
  Step 6: If the middle element is greater than the target value, update the high pointer to (middle index – 1) and go to step 3.
  Step 7: If the middle element is less than the target value, update the low pointer to (middle index + 1) and go to step 3.
  Step 8: Repeat steps 3-7 until the target value is found and the low pointer becomes greater than the high pointer.
  Step 9: If the low pointer is greater than the high pointer and the target value is not present in the array then return -1.

- *Program:* Here is an implementation of the Binary search algorithm in C language.

```
#include <stdio.h>
int main()
{
   int n, ns;
printf("Enter the number of elements in the array: ");
scanf("%d", &n);
  int arr[n];
printf("Enter %d sorted elements:\n", n);
  for (int i = 0; i< n; i++)
scanf("%d", &arr[i]);
printf("Enter the element to be searched: ");
scanf("%d", &ns);
int low = 0, high = n - 1, mid, found = 0;
 while (low <= high)
{
      mid = (high + low) / 2;
      if (arr[mid] == ns)
      {
```

```
printf("Element found at position %d.\n", mid);
            found = 1;
            break;
        }
else if (arr[mid] < ns)
            low = mid + 1;
       else
            high = mid - 1;
}
   if (found==0)
printf("Element not found in the array.\n");
   return 0;
}
```

C. Complexity in searching algorithms

The time and space complexity of the searching algorithms is commonly represented using Big O notation, which describes the upper bounds of an algorithm's growth rate as a function of n, which is the number of elements in the given list, as shown in TABLE II.

TABLE II: Time and space complexity of Searching algorithms.

ALGORITHMS	TIME COMPLEXITY			SPACE COMPLEXITY		
	Best Case	**Average Case**	**Worst Case**	**Best Case**	**Average Case**	**Worst Case**
LINEAR SEARCH [3]	O(1)	O(n)	O(n)	O(1)	O(1)	O(1)
BINARY SEARCH [3]	O(1)	O(logn)	O(logn)	O(1)	O(1)	O(1)

IV. CONCLUSIONS

In conclusion, this research paper offers a comprehensive and unique exploration of Searching and Sorting algorithms, distinguishing itself through several key features. Unlike other publications, our paper not only provides a concise introduction but also delves into the properties of these algorithms, offering readers a deeper understanding of their underlying principles. Furthermore, our inclusion of pictorial representations of algorithmic workings serves to enhance clarity and facilitate comprehension, a feature often overlooked in similar studies.

Furthermore, our paper distinguishes itself through the inclusion of algorithmic implementations presented alongside C code, facilitating practical understanding and direct application of the discussed concepts. Moreover, our meticulous examination of comparative time and space complexities provides valuable insights into the efficiency and performance trade-offs inherent in various algorithmic methodologies.

In summary, this research paper excels in its comprehensive approach, integrating theoretical understanding with practical implementations and thorough comparative analyses. We anticipate that this paper will prove to be an invaluable asset for researchers, students, and practitioners, promoting a deeper comprehension and appreciation of searching and sorting algorithms within the field of computer science and beyond.

REFERENCES

[1] R. Renu and M. Manisha, "Mq sort an innovative algorithm using quick sort and merge sort", *International Journal of Computer Applications*, 2015.

[2] V. P. Parmar and C. Kumbharana, "Comparing linear search and binary search algorithms to search an element from a linear list implemented through static array dynamic array and linked list", *International Journal of Computer Applications*, 2015.

[3] A. E. Jacob, N. Ashodariya and A. Dhongade, "Hybrid search algorithm: Combined linear and binary search algorithm", *2017 International Conference on Energy Communication Data Analytics and Soft Computing (ICECDS)*, 2017.

[4] Y. Chauhan and A. Duggal, "Different sorting algorithms comparison based upon the time complexity", *Int. J. Res. Anal. Rev*, 2020.

[5] S. Buradagunta, J. D. Bodapati, N. B. Mundukur and S. Salma, "Performance comparison of sorting algorithms with random numbers as inputs", *Ingénierie des Systèmesd'Information*, 2020.
[6] F. A. Agha and H. Nawaz, "Comparison of bubble and insertion sort in rust and python language", *Internat. l J*, 2021.
[7] "Binary search algorithm", *MyGreatLearning*, 2021, [online] Available: https://www.mygreatlearning.com/blog/binary-search-algorithm/.
[8] "Merge sort", *Studytonight*, 2021, [online] Available: https://www.studytonight.com/data-structures/merge-sort.
[9] "Quick sort", *Programiz*, 2021, [online] Available: https://www.programiz.com/dsa/quick-sort.
[10] A. Salihu, M. Hoti and A. Hoti, "A review of performance and complexity on sorting algorithms", December 2022.
[11] You Yang, Ping Yu and Yan Gan, "Experimental study on the five sort algorithms", 2011 Second International Conference on Mechanic Automation and Control Engineering, 15-17 July 2011.
[12] Asha Elza Jacob, Nikhil Ashodariya and Aakash Dhongade, "Hybrid search algorithm: Combined linear and binary search algorithm", 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS), 01-02 August 2017.
[13] Najma Sultana, Smita Paira, Sourabh Chandra and S. K. Safikul Alam, "A brief study and analysis of different searching algorithms", 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), 22-24 February 2017.
[14] Marcellino Marcellino, Davin William Pratama, Steven Santoso Suntiarko and Kristien Margi, "Comparative of Advanced Sorting Algorithms (Quick Sort, Heap Sort, Merge Sort, Intro Sort, Radix Sort) Based on Time and Memory Usage", 2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI), 28-28 October 2021.
[15] Vanam Chandana, Guvvala Tejaswini, Lalita Gupta and R.N. Yadav, "Algorithm Development Analysis: searching and sorting",2022 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS), 19-20 February 2022.
[16] Thomas H. Cormen, "Algorithms for Sorting and Searching", MIT Press, 2013.
[17] Wang Min, " Analysis on Bubble Sort Algorithm Optimization", 2010 International Forum on Information Technology and Applications, 16-18 July 2010.
[18] Tithi Paul, "Enhancement of Bubble and Insertion Sort Algorithm Using Block Partitioning", 2022 25th International Conference on Computer and Information Technology (ICCIT), 17-19 December 2022.