**Articles and Books Managing Project**

|  |  |
| --- | --- |
| Project Supervisor | Mr. Samyan Qayyum Wahla |

|  |  |
| --- | --- |
| Group Number | CS261F22PID25 |
| Group Members | Muhammad Nazir(2021-CS-8)  Subhan Anjum(2021-CS-13)  Muhammad Umar(2021-CS-24) |

**Table of Content:**

[1 Overview: 4](#_Toc119573154)

[1.1 Project description 4](#_Toc119573155)

[1.2 The end user of the project 4](#_Toc119573156)

[1.3 Motivation for Project 4](#_Toc119573157)

[2 Technical Details: 4](#_Toc119573158)

[2.1 Sorting Algorithms 4](#_Toc119573159)

[2.1.1 Insertion sort 4](#_Toc119573160)

[2.1.1.1 Dry Run: 4](#_Toc119573161)

[2.1.1.2 Time Complexity: 1](#_Toc119573162)

[2.1.2 Bubble Sort 1](#_Toc119573163)

[2.1.2.1 Dry Run: 1](#_Toc119573164)

[2.1.2.2 Time Complexity 1](#_Toc119573165)

[2.1.3 Selection Sort 1](#_Toc119573166)

[2.1.3.1 Dry Run: 2](#_Toc119573167)

[2.1.3.2 Time Complexity: 2](#_Toc119573168)

[2.1.4 Merge Sort 2](#_Toc119573169)

[2.1.4.1 Dry Run: 2](#_Toc119573170)

[2.1.5 Hybrid Merge Sort 3](#_Toc119573171)

[2.1.5.1 Dry Run: 3](#_Toc119573172)

[2.1.5.2 Time Complexity: 3](#_Toc119573173)

[2.1.6 Radix Sort 3](#_Toc119573174)

[2.1.6.1 Dry Run: 4](#_Toc119573175)

[2.1.6.2 Time Complexity: 4](#_Toc119573176)

[2.1.7 Counting Sort 4](#_Toc119573177)

[2.1.7.1 Dry Run: 4](#_Toc119573178)

[2.1.7.2 Time Complexity: 4](#_Toc119573179)

[2.1.8 Bucket Sort 4](#_Toc119573180)

[2.1.8.2 Time Complexity: 4](#_Toc119573181)

[2.1.9 Quick Sort 5](#_Toc119573182)

[2.1.9.1 Dry Run: 5](#_Toc119573183)

[2.1.9.2 Time Complexity: 5](#_Toc119573184)

[2.1.10 Heap Sort 5](#_Toc119573185)

[2.1.10.1 Dry Run: 5](#_Toc119573186)

[2.1.10.2 Time Complexity: 5](#_Toc119573187)

[2.1.11 Shell Sort 6](#_Toc119573188)

[2.1.11.1 Dry Run: 6](#_Toc119573189)

[2.1.11.2 Time Complexity: 6](#_Toc119573190)

[2.1.12 Comb Sort 6](#_Toc119573191)

[2.1.12.1 Dry Run: 6](#_Toc119573192)

[2.1.12.2 Time Complexity: 6](#_Toc119573193)

[2.1.13 Brick Sort 7](#_Toc119573194)

[2.1.13.1 Dry Run: 7](#_Toc119573195)

[2.1.13.2 Time Complexity: 7](#_Toc119573196)

[2.2 Searching Algorithms 7](#_Toc119573197)

[2.2.1 Linear Search 7](#_Toc119573198)

[2.2.1.1 Dry Run: 7](#_Toc119573199)

[2.2.1.2 Time Complexity: 7](#_Toc119573200)

[2.2.2 Boolean Search 7](#_Toc119573201)

[2.2.2.1 Dry Run: 7](#_Toc119573202)

[2.2.2.2 Time Complexity: 8](#_Toc119573203)

[2.2.3 Binary Search 8](#_Toc119573204)

[2.2.3.1 Dry Run: 8](#_Toc119573205)

[2.2.3.2 Time Complexity: 8](#_Toc119573206)

[3 Main Ui Window 8](#_Toc119573207)

[3.1 UI Elements 9](#_Toc119573208)

[3.1.1 Start button 9](#_Toc119573209)

[3.1.2 Stop button 9](#_Toc119573210)

[3.1.3 Pause button 9](#_Toc119573211)

[3.1.4 Resume button 10](#_Toc119573212)

[3.1.5 Table Widget 10](#_Toc119573213)

[3.1.6 Progress Bar 10](#_Toc119573214)

[3.1.7 Time Label 10](#_Toc119573215)

[3.2 Sorting Tab 11](#_Toc119573216)

[3.3 Searching Window 11](#_Toc119573217)

[3.4 Scrapping Window 12](#_Toc119573218)

**Table of Figures:**

[Main Window 1 9](#_Toc117885494)

[Sorting Window 1 11](file:///C:\Users\Muhammad%20Nazir\Desktop\subhan.docx#_Toc117885524)

[Searching Window 1 12](#_Toc117885537)

[Scrapping Window 1 12](#_Toc117885547)

# **1 Overview:**

# 

## **1.1 Project description**

The System eliminates the chance of errors when performing its tasks, considerably reducing manual labor and enabling a smooth flow of searching and sorting activities. In this project, users would be able to sort, search and scrap the data (of the website). The end users of an Articles and Books managing project are usually people for whom libraries are intended (i.e. pupils, students, researchers, senior users, library staff, etc.).

## **1.2 The end user of the project**

The end users of an Articles and Books managing project are usually people for whom libraries are intended (i.e. pupils, students, researchers, senior users, library staff, etc.).

## **1.3 Motivation for Project**

In libraries, we have to face many difficulties in searching for books. We choose the Article and books management project so that it will be helpful both for students and library managers to keep constant track of all the books available in the library. It allows both the admin and the student to search for the desired book. Our project will affect the operational level and strategic levels. It will be helpful for students and librarians. They can use sorting algorithms and searching algorithms which can make their life helpful.

# **2 Technical Details:**

## **2.1 Sorting Algorithms**

We will be doing 10-12 sorting algorithms whose description is given below:

### **2.1.1 Insertion sort**

In insertion sort we select second element as key. Then we compare it with previous elements of array. If key is smaller than any previous element than we take it to that element position and the key’s previous elements are moved towards next indexes. The process goes on and we select key for each iteration and at last the unsorted array becomes sorted.

#### **2.1.1.1 Dry Run:**

**Key = 43**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | **43** | 76 | 2 | 98 |

**Key = 76**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 43 | 76 | 2 | 98 |

**Key = 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 43 | 76 | 2 | 98 |

**Sorted Array:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 5 | 43 | 76 | 98 |

#### **2.1.1.2 Time Complexity:**

Best Case: O(n) Average: θ (n2) Worst case: O(n2)

### 

### **2.1.2 Bubble Sort**

In this sorting, first element is already sorted so we start from second element. We compare it with previous element and if it is smaller than previous element than previous element is stored in some temporary variable from which we can access it and the selected element takes the position of previous elements. In short, we swap them and process goes on and we get sorted array.

#### **2.1.2.1 Dry Run:**

Unsorted Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 9 | 8 | 7 | 6 | 4 |

First element is already sorted so we start from j=2

At j = 2

A[ j-1 ] > A[ j ]

So, temp = A[ j-1 ] = 9

A[j-1] = A[j] = 8

A[j] = 9

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 9 | 8 | 7 | 6 | 4 |

Sorted part:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 9 | 7 | 6 | 4 |

At j = 3

A[3] = 7

Hence it shifts to previous positions because its previous elements are greater than it.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 9 | 7 | 6 | 4 |

#### **2.1.2.2 Time Complexity**

Best Case: Ω(n) Average: θ (n2) Worst case: O(n2)

Sorted Part:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | 8 | 9 | 6 | 4 |

At j = 4

A[4] = 6

Hence it shifts to previous positions because its previous elements are greater than it.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 9 | 7 | 6 | 4 |

Sorted part:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 7 | 8 | 9 | 4 |

At j = 5

A[5] = 4

Hence it shifts to previous positions because its previous elements are greater than it. We move it to first location and moves forward other elements.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 9 | 7 | 6 | 4 |

Sorted Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 6 | 7 | 8 | 9 |

### **2.1.3 Selection Sort**

In selection sort, first smallest element is finding from the complete array. We compare it with previous elements and take it to the starts position of array. After first element, then second smallest element is selected from unsorted part of array and this process goes on and we get sorted array.

#### **2.1.3.1 Dry Run:**

Unsorted Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | -1 | 2 | 3 | 5 |

i=1

minimum-index=1

j=i+1=2

a[minimum-index] =a[i]=4 that is greater than a[j]=a[2] =-1

so, we swap them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | -1 | 2 | 3 | 5 |

Sorted part:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -1 | 4 | 2 | 3 | 5 |

i=2

minimum-index=1

j=i+1=3

a[minimum-index] =a[i]=4 that is greater than a[j]=a[3] =2

so we swap them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -1 | 4 | 2 | 3 | 5 |

Sorted part;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -1 | 2 | 4 | 3 | 5 |

i=3

minimum-index=1

j=i+1=4

a[minimum-index] =a[i]=4 that is greater than a[j]=a[4] =3

so, we swap them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -1 | 2 | 4 | 3 | 5 |

Sorted part;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -1 | 2 | 3 | 4 | 5 |

i=4

minimum-index=1

j=i+1=5

a[minimum-index] =a[i]=4 that is smaller than a[j]=a[5] =5

so, we do not swap them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -1 | 2 | 3 | 4 | 5 |

Fully Sorted Array

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -1 | 2 | 3 | 4 | 5 |

#### **2.1.3.2 Time Complexity:**

Best Case: Ω(n2) Average: θ (n2) Worst case: O(n2)

### **2.1.4 Merge Sort**

In this sorting, we use divide and conquer rule. Given array is divided into sub parts and then sub parts are further divided into smallest part. At last we get sub parts of one element. After completion of divide method, elements start combining. In this way we get sorted array.

#### **2.1.4.1 Dry Run:**

Unsorted Array:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 54 | 2 | 72 | 23 | 12 | 32 | 76 | 12 |

23

12

32

76

12

72

2

54

1

3

32

12

23

2

72

54

3

1

76

12

76

12

32

12

23

72

2

54

3

1

12

23

3

1

Now after partition, these elements start combining.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 12 | 12 | 23 | 32 | 54 | 72 | 76 |

23

12

76

32

23

12

12

3

72

544

1

2

3

1

2

1

3

72

54

23

32

76

12

12

### **2.1.5 Hybrid Merge Sort**

In hybrid merge sort, we use merge sort and insertion sort. Insertion sort is best for arrays of short lengths and merge is best for arrays of long length. This algorithm is efficient and sorts elements in short time as compared to insertion, bubble and selection and many more.

#### **2.1.5.1 Dry Run:**

In this algorithm we use insertion sort and merge sort. We use insertion sort for small array, when size of array is enough large that insertion sort starts taking time more than merge sort than we use merge sort. We can say that this sort is a combination of merge and insertion sort.

#### **2.1.5.2 Time Complexity:**

Best Case: Ω(n lgn) Average: θ(n lgn) Worst case: O(n lgn)

### **2.1.6 Radix Sort**

It is a linear sorting algorithm in which we sort elements of arrays by comparing their digits. First, very right digit of every elements of array is compared with each other and sort elements with respect to very right digit. This process repeats for further digits and hence array becomes sorted.

#### **2.1.6.1 Dry Run:**

Unsorted Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 89 | 231 | 12 | 100 | 90 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 100 | 90 | 231 | 12 | 89 |
| 100 | 12 | 231 | 89 | 90 |

Sorted Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 89 | 90 | 100 | 231 |

#### **2.1.6.2 Time Complexity:**

O(d\*(n+b))

### **2.1.7 Counting Sort**

In this sorting algorithm, we find maximum element and then create array of (maximum+1) length. Then we count the existence of element and stored this count in new array. After date we take cumulative sum of elements and count array is formed. Than we compare the elements of array and subtract the index of given array from count array and stored the element of arrray at that position.

#### **2.1.7.1 Dry Run:**

Unsorted Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 5 | 2 | 2 | 1 |

Find max element

Initialize an array of length max+1 with all the elements 0 used for storing the count

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 0 | 0 | 0 | 0 | 0 | 0 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |

Now store the count of each element at their respective index

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 0 | 1 | 2 | 0 | 0 | 2 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |

Now store the cumulative sum of elements of the count array

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 0 | 1 | 3 | 3 | 3 | 5 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |

Find the index of each element of the original array in the count array

Given Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 5 | 5 | 2 | 2 | 1 |

Count Array:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 0 | 1 | 3 | 3 | 3 | 5 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |

**5-1=4** and same for other elements

Sorted Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 2 | 5 | 5 |

#### **2.1.7.2 Time Complexity:**

Best Case: Ω(n +k) Average: θ(n +k) Worst case: O(n +k)

### **2.1.8 Bucket Sort**

It divides items into multiple groups called buckets. Bucket sort items are first divided evenly into groups called buckets and then sorted using other sorting algorithms. Then sort and collect the elements.

#### **2.1.8.2 Time Complexity:**

Best Case: Ω(n +k) Average: θ(n +k) Worst case: O(n2)

### **2.1.9 Quick Sort**

In this sorting, we use divide and conquer rule. We select one element as pivot and partition is done on left (Smaller than pivot) and right (Larger than pivot) side of pivot. Then again, we choose pivot element from small part of array and divide it in left and right part.

#### **2.1.9.1 Dry Run:**

Pivot=10

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 5 | 20 | 15 | 10 |

There is partition around pivot 10.

Partition on left side of pivot. Partition on right side:

New Pivot=5 New pivot=15

|  |  |
| --- | --- |
| 3 | 5 |

|  |  |
| --- | --- |
| 20 | 15 |

3

empty

15

empty

Sorted Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 5 | 10 | 15 | 20 |

#### **2.1.9.2 Time Complexity:**

Best Case: Ω(n log(n)) Average: θ(n log(n)) Worst case: O(n2)

### **2.1.10 Heap Sort**

In this algorithm, we use max heapify function. We compare the elements in such a way. The child elements are compared with parent elements. And if child element is greater than parent element than moves it to parent element. Hence the array is sorted. We can also use min heapify for sorting array in decreasing order.

#### **2.1.10.1 Dry Run:**

Unsorted Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 14 | 13 | 17 | 11 | 18 | 15 |

18

14

17

14

13

17

15

13

11

15

18

11

#### **2.1.10.2 Time Complexity:**

Best Case: Ω(n log(n)) Average: θ(n log(n)) Worst case: O(n log(n))

### **2.1.11 Shell Sort**

In Shell Sort, we divide the length of array by 2 and then compare the elements in which gap is equal to our result(n/2). Then we divide original length by 4 and so on by 8 and hence we get the sorted array.

#### **2.1.11.1 Dry Run:**

Unsorted Array:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 8 | 3 | 7 | 5 | 6 | 4 | 1 |

The length of the array is 8 so 8/2 =4 elements at the interval are compared and swapped

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 8 | 3 | 7 | 9 | 6 | 4 | 1 |

According to it next elements will also swap and process goes on and we get

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 6 | 3 | 1 | 9 | 8 | 4 | 7 |

In the second loop an interval of n/4=8/4=2 is taken and these are also swapped as above according to smaller element.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 6 | 3 | 1 | 9 | 8 | 4 | 7 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 6 | 5 | 1 | 9 | 8 | 4 | 7 |

And finally, we get

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 6 | 5 | 7 | 9 | 8 |

And now the interval is n/8=8/8=1 . In this case the 0 and 1 ,1 and 2 and so on are compared and swapped if required.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 6 | 5 | 7 | 9 | 8 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 6 | 5 | 7 | 9 | 8 |

And now they are compared and swapped and we get the sorted array

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

#### **2.1.11.2 Time Complexity:**

Best Case: Ω(n log(n)) Average: θ(n log(n)) Worst case: O(n2)

### **2.1.12 Comb Sort**

In Comb Sort, we divide the length of array by 1.3 and then we take gap of elements and compare elements in which the gap is equal to our result and then result is further divided by 1.3 and then compare elements and swapped them so we get sorted array.

#### **2.1.12.1 Dry Run:**

Unsorted Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | 5 | 20 | 15 | 3 |

Gap Value= 5/1.3=3 . We compare 10 with 15 and 5 with 3 then swap 5 and 3them.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | 3 | 20 | 15 | 5 |

Gap Value= 3/1.3=2 .We compare 10 with 20, 3 with 15 and 20 .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | 3 | 5 | 15 | 20 |

Gap Value= 2/1.3=1 . We compare each corresponding element and swap them if required.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 10 | 5 | 15 | 20 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 5 | 10 | 15 | 20 |

Sorted Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 5 | 10 | 15 | 20 |

#### **2.1.12.2 Time Complexity:**

Best Case: O(n log(n)) Average: O(n2/2p) Worst case: O(n2)

### **2.1.13 Brick Sort**

In this sorting in the first phase elements are compared as a pair with each other. And if element is smaller than its previous element than swap it. In second phase besides first and last element, the remaining array is divided into pairs and then swap them .this process goes on respectively and we get sorted array.

#### **2.1.13.1 Dry Run:**

Unsorted Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 2 | 4 | 6 | 5 | 1 |

In the first phase they are compared as a pair with each other.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 2 | 4 | 6 | 5 | 1 |

And they will be swapped

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 6 | 1 | 5 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 1 | 6 | 5 |

Again, they are swapped and sorted again

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 3 | 1 | 4 | 5 | 6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 1 | 3 | 4 | 5 | 6 |

And now we will get a sorted array by swapping according to smaller elements

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 |

#### **2.1.13.2 Time Complexity:**

O(N2)

## **2.2 Searching Algorithms**

### **2.2.1 Linear Search**

In this algorithm, we start finding element from first index of array to last index and compare the element. Where the element is matched with array element, we return index and element.

#### **2.2.1.1 Dry Run:**

Find 20

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 11 | 12 | 13 | 14 | 20 | 21 | 18 |

#### **2.2.1.2 Time Complexity:**

O(n)

### **2.2.2 Boolean Search**

In Boolean search, we use AND, OR and NOT operators to combine different arguments. We get results if the required operators are true.

#### **2.2.2.1 Dry Run:**

In Boolean search, we use AND, OR and NOT operators. In fact, it is some kind of linear search in which we search elements of different columns by joining them with logic operators. For And both arguments should be true. For OR operator, any one can be true. For NOT operator that element is searched which is not written in search box.

#### **2.2.2.2 Time Complexity:**

O(1)

### **2.2.3 Binary Search**

In binary search, we compare the element with middle element. Array is divided into two parts and that part in which element is not present is excluded and we find element in remaining array. The remaining array in also divided by finding mid and one unwanted part get excluded and in tis way element is searched.

#### **2.2.3.1 Dry Run:**

Find 5:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Index | L=0 | 1 | 2 | M=3 | 4 | 5 | H=6 |

As searching element is greater than mid element (i.e. 5>4) so we now search next index of mid to last element. We find M=(L+ H)/2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Index | L=0 | 1 | 2 | 3 | L=4 | M=5 | H=6 |

As searching element is smaller than mid element (i.e. 5<6) so we now search in remaining part.

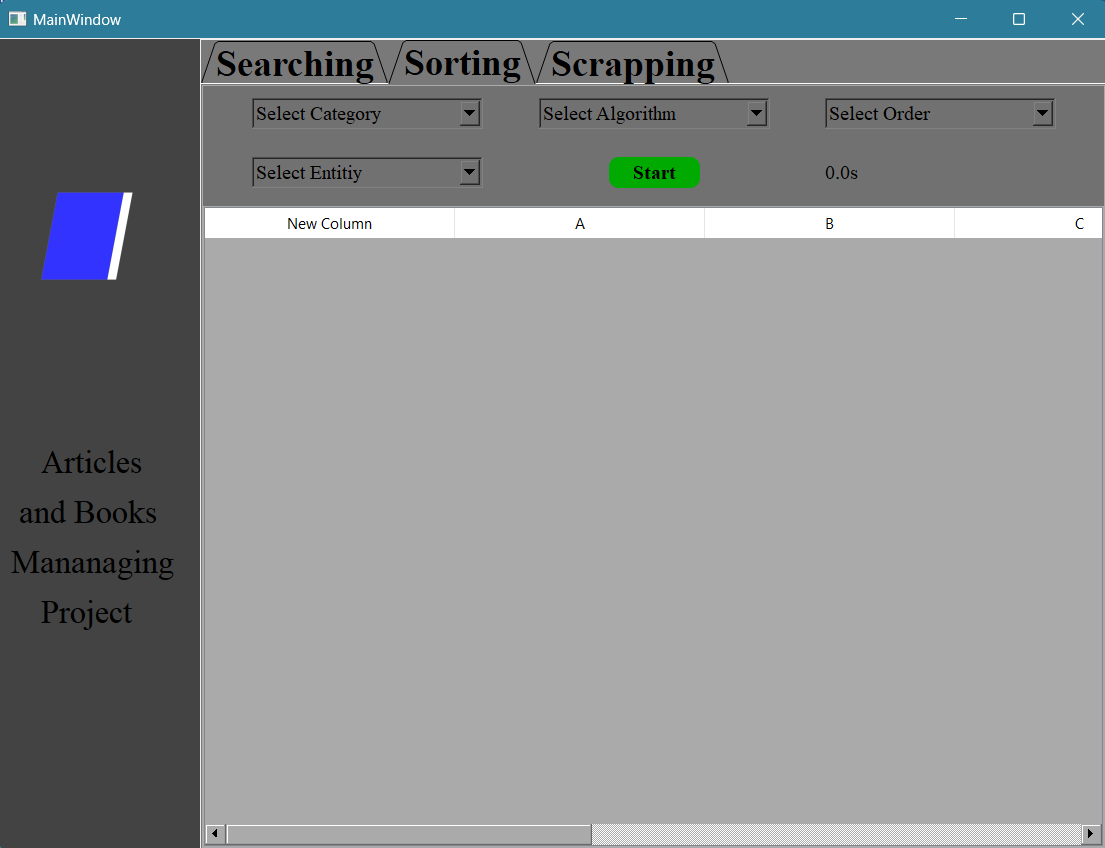
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Index | L=0 | 1 | 2 | 3 | L=M=H=4 | 5 | 6 |

Hence, we searched 5.

#### **2.2.3.2 Time Complexity:**

O (log n).

# **3 Main Ui Window**



Main Window 1

## **3.1 UI Elements**

### **3.1.1 Start button**

****

This button will work when you have selected all the values from the checkbox either in sorting or searching and also in scrapping.

### **3.1.2 Stop button**

****

This button is used to stop scrapping while doing it.

### **3.1.3 Pause button**

****

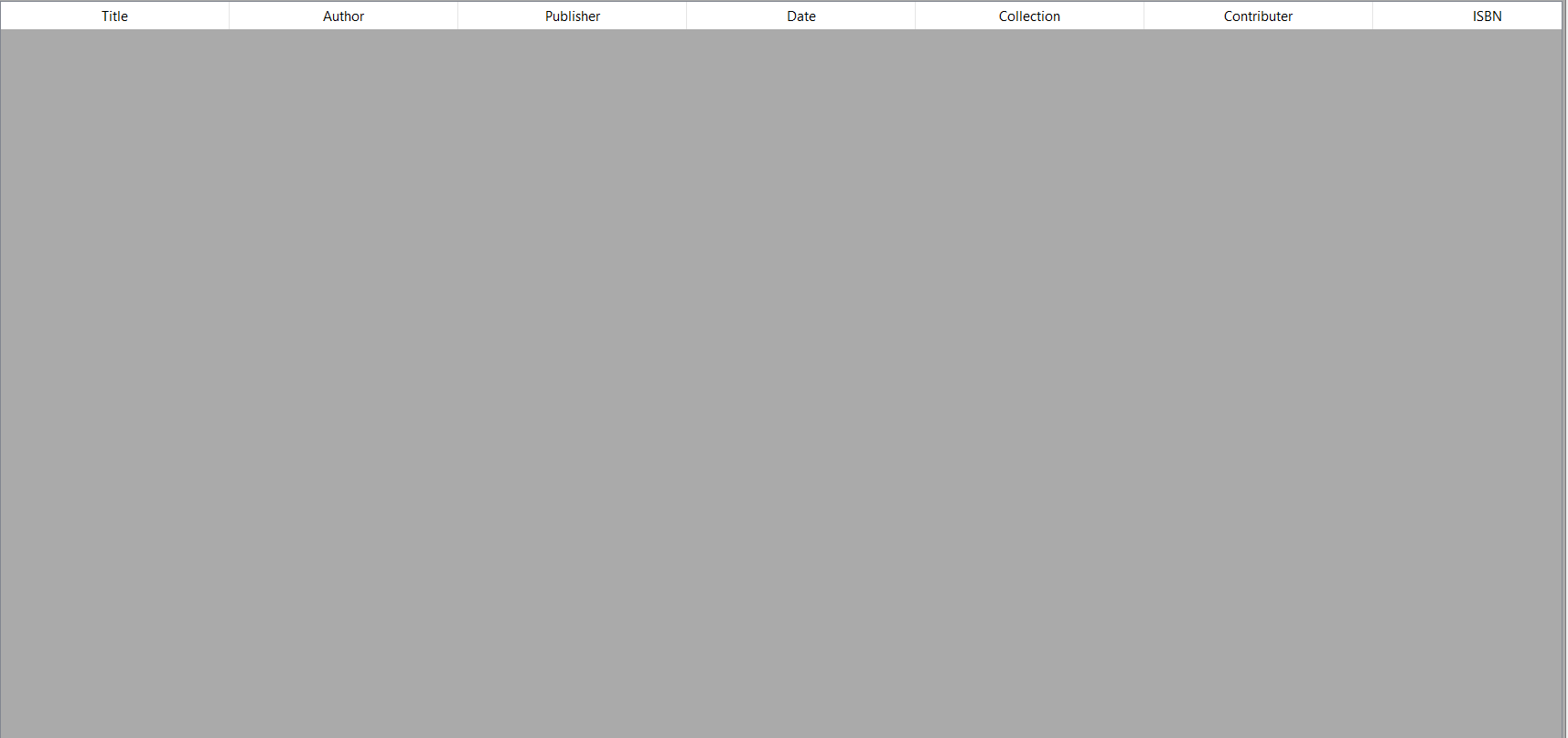
This button will allow the user to pause the scrapping of books and articles whenever he wants.

### **3.1.4 Resume button**

****

This button will allow the user to resume scrapping the books and articles whenever he wants.

### **3.1.5 Table Widget**

****

This Table widget will show the required data to the user according to the conditions given by him.

### **3.1.6 Progress Bar**

****

This progress bar will show the amount of data scrapped and the percentage will also be shown on the side of it.

### **3.1.7 Time Label**



This will show you the time required to sort, scrap or search the given data.

## **3.2 Sorting Tab**

Sorting Window 1

This is how we use sorting algorithms on Books and also show the time taken by it to the user of the project.

## **3.3 Searching Window**

This is how we use a search algorithm. It has 3 filters: starts with, contains, and ends with and the user gets the desired output.

****

Searching Window 1

## **3.4 Scrapping Window**

****

Scrapping Window 1

In this window, the user will be able to scrap the data of books and articles the user will be able to start, stop and resume whenever he wants.