**Data Structures and Algorithms**

CS261-Mid Project Final Report



|  |  |  |
| --- | --- | --- |
| **Group Id** | G12 | |
| **Registration No.** | 2020-CS-124 | 2020-CS-150 |
| **Project Name** | The Book Town | |
| **Project Supervisor** | Mr. Samyan Qayyum Wahla | |

Department of Computer Science

University of Engineering and Technology, Lahore

Pakistan

Table of Contents

[Project Details 2](#_Toc87026339)

[Business Case 3](#_Toc87026340)

[Scrapping Details 3](#_Toc87026341)

[UI Details 4](#_Toc87026342)

[Algorithms 5](#_Toc87026343)

[**Description Of Algorithms** 6](#_Toc87026344)

[**Working of Sorting Algorithms** 7](#_Toc87026345)

[**Sorting Filters:** 27](#_Toc87026346)

[**Searching Filters:** 27](#_Toc87026347)

[**Multi-Level Sorting:** 27](#_Toc87026348)

[Collaboration 27](#_Toc87026349)

[Task Division 28](#_Toc87026350)

# Project Details

**Project Title:** The BookTown

**Project Description:**

This project is based on books. You can say that is it a digital library. It will provide a user-friendly environment in which a user can search a book of his choice. A book can be search by its keywords, title, author, ISBN or publisher. It will contain millions of books of different categories.

There are mainly three categories:

1. Adult
2. Children
3. Academic

The subcategories of books are science books, novels, religion books, cooking books, coloring books and literature etc. This collection of books will be gathered by **scrapping**. The books will be shown with their information i.e., title, author, category, price, pages, ISBN, publisher.

In this project, at least 1 million books will be scrapped. It will contain the options to pause, start, resume and stop, with the progress bar showing the number of books scrapped. Attributes of entity will be at least 7 or more. For desktop application UI has the option to take URL as input and scrap. UI will be designed using **PyQT**. And for scrapping we will use **BeautifulSoap** One page will display the list of chosen books.

UI will have the option for sorting of each column. A column can be sort alphabetically (a-z or z-a) and by price range. It will display scrapping time in milliseconds. UI will have the option for searching based on each column. A book can be search by alphabetically, and by price. It also contains multi-level sorting in which one column is sort according to the second column. Advanced filters for string columns will be implemented such as contains, end with, starts with etc. User will have option to sort multiple columns.

**Project Motivation:**

The motivation of this project is to deliver source of knowledge in a convenient way. It helps user in search of a specific book of a specific author. It is a great platform for book readers as they are provided millions of books and they can choose any book of their choice.

# Business Case

**Business Need:**

The problem is that it is very difficult for a person to search a single book in millions of books. This project is a benefit for Educational System and Library Management System. It provides user-friendly environment.

**User’s Benefit:**

This project is for readers. Those people whose passion is to gather knowledge from books and are interested in reading every kind of book.

**Impact on society:**

This project will have a positive impact in society as it is an ocean of knowledge. If this project is implemented so it will be a great source of knowledge for book readers. Else book readers will be deprived of convenient way of searching book and have trouble in finding their desired books.

# Scrapping Details

**Name of Entities:**

* Title
* Author
* Category
* Price
* Pages
* ISBN
* Publisher

**Scrapping Website:**

<https://www.readings.com.pk>

**Sample:**

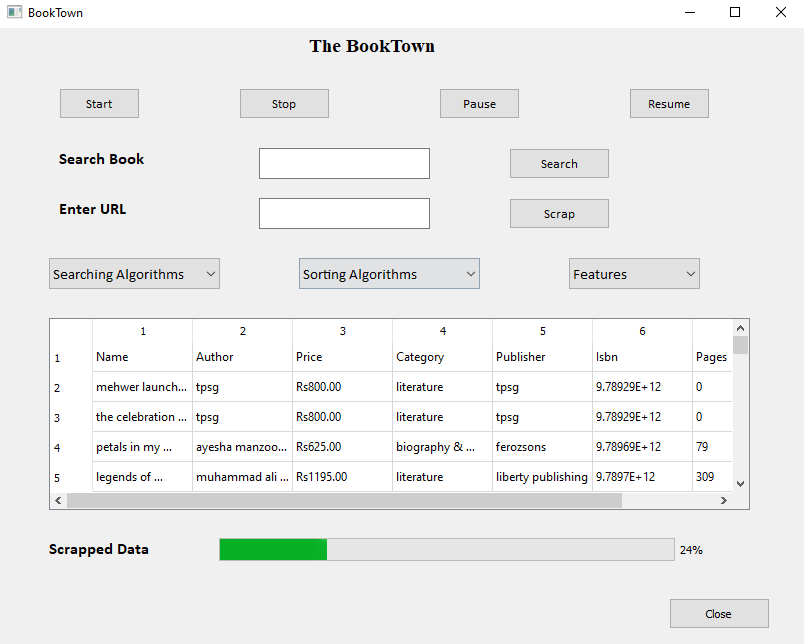
**

# UI Details

**UI components:**

|  |  |  |
| --- | --- | --- |
| Name | Text Field | Description |
| Start | Button | It is a used to start scrapping |
| Stop | Button | It is used to stop scrapping. |
| Pause | Button | It is used to pause scrapping. |
| Resume | Button | It is used to resume scrapping. |
| Search Book | Text Field | It is the text field to search a book by its searching algorithms. |
| Enter URL | Text Field | It is used to enter the URL and scrap its data. |
| Search | Button | This button is used to search the book. |
| Scrap | Button | This button is used to scrap data. |
| Searching Algorithms | Combo Box | This combo box contains searching algorithms. |
| Sorting Algorithms | Combo Box | This combo box contains sorting algorithms. |
| Features | Combo Box | It contains features like how data will be sort. |
| Scrapped Data | Progress Bar | It is used to show that how much data has been scrapped. |
| Close | Button | It is used to exit. |

**UI in PyQt5:**

****

# Algorithms

**Sorting Algorithm Names:**

* Insertion Sort
* Merge Sort
* Bubble Sort
* Selection Sort
* Quick Sort
* Hybrid Sort
* Counting Sort
* Radix Sort
* Bucket Sort
* Tim Sort

**Searching Algorithm Names:**

* Linear Search
* Binary Search

## **Description Of Algorithms**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Algorithm Name** | **Description** | | Insertion Sort | Insertion Sort is the simplest algorithm in which an element is inserted at its right place and this process goes on until all the elements of an array is sorted. | | Merge Sort | Merge sort requires divide and conquer rule. In this sorting algorithm an array is divided into two parts and then it gets subdivided until a single element remains. After that it will be sort according to the requirement (increasing or decreasing). | | Bubble Sort | In bubble sort, it compares two adjacent indexes and checks if it is in order or not. It then swaps two elements if they aren’t in order and this process continuous until the array is sorted. | | Selection Sort | It first finds the smallest element and then second smallest elements and so on. So, this is how selection sort sorts an array. | | Quick Sort | It chooses a pivot and place it in its position. All the elements greater than pivot are placed at right side and all the elements smaller than pivot are placed on left side. Then it executes recursively. | | Counting Sort | This algorithm sorts an array without comparison. It places number of occurrences in a new array. And it sorts the input array according to the values of occurrences. | | Radix Sort | Radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. | | Bucket Sort | Bucket sort, or bin sort, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. | | Tim Sort | Tim sort is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. | | Hybrid Sort | A Hybrid Algorithm is an algorithm that combines two or more other algorithms that solve the same problem, either choosing one (depending on the data) or switching between them throughout the algorithm. | | Linear Search | In a linear search or sequential search, an element is found sequentially in a list. Just like array, it goes through every element until the required element is found. | | Binary Search | Binary search uses *divide and conquer rule.* It first divides the array into two equal parts and then search the right and left array individually. | |

## **Working of Sorting Algorithms**

|  |  |
| --- | --- |
| ***Insertion Sort*** |  |
| Description | Insertion Sort is the simplest **comparison-based** algorithm in which an element is inserted at its right place and this process goes on until all the elements of an array is sorted.  First of all, a key is taken which is initially second index of array. It is because first element is always sorted. Then that key has to be placed at its right place. When that key is placed at its right place, key is iterated and then that value is to be placed at its place. This process will continue till array is sorted.  Now, consider you have some pages in your hand which is in unsorted form. The pages are numbered from 1 to 10. Now at first let’s say you have a page 3(arr[0]), that is already sorted. The page placed at second number is 2 i.e., key, so you will place second page at first and then iterate the value of key which will be the page placed at arr[2]. This is how insertion sort works. |
| Pseudo Code | INSERTION-SORT (A)     for j = 2 to length[A]             do key = A[j]              i = j – 1              while i > 0 and A[i] > key                        do A [i + 1] = A[i]                         i = i - 1              A [i + 1] = key |
| Python Code | def InsertionSort(arr):     for j in range (1, len(arr)):         key = arr[j]         i = j - 1         while i >= 0 and arr[i] > key:             arr[i+1] = arr[i]             i = i - 1         arr[i+1] = key  arr = [] n = int (input ("Enter number of elements: ")) for m in range(n):     num = int (input ())     arr.append(num) InsertionSort(arr) print(arr) |
| Time Complexity | |  |  |  | | --- | --- | --- | | Worst Case | Average Case | Best Case | | O (n2) | O (n2) | O (n) | |
| Proof of Correctness | **Initialization:**  We start by showing that the loop invariant holds before the first loop iteration, when j = 21 The subarray A[1..j..-1], therefore, consists of just the single element A[1], which is in fact the original element in A[1]. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.  **Maintenance:**  Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the for loop works by moving A[j-1], A[j-2], A[j-3], and so on by one position to the right until it finds the proper position for A[j], at which point it inserts the value of A[j]. The subarray A [1....j] then consists of the elements originally in A [1.... j], but in sorted order. Incrementing j for the next iteration of the for loop then preserves the loop invariant. A more formal treatment of the second property would require us to state and show a loop invariant for the while loop. At this point, however, we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.  **Termination:**  Finally, we examine what happens when the loop terminates. The condition causing the for loop to terminate is that j > A.length. Because each loop iteration increases j by 1, we must have j = n+1 at that time. Substituting n+1 for j in the wording of loop invariant, we have that the subarray A [1...n] consists of the elements originally in A[1...n], but in sorted order. Observing that the subarray A [1.. n] is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct. |
| Strengths | 1. It is efficient for smaller number of inputs. 2. It only requires a constant amount of additional memory space i.e., O (1). 3. It has a fairly fast running case when the input is nearly sorted. |
| Weaknesses | 1. It is not efficient for larger amount of data. 2. Time complexity of insertion sort is O(n2) which requires n2 steps for sorting N element. 3. For unsorted/reverse-sorted data, it is slow. |
| Dry Run |  |

|  |  |
| --- | --- |
| ***Merge Sort*** |  |
| Description | Merge sort is a **comparison-based** sorting algorithm which requires divide and conquer rule. In this sorting algorithm an array is divided into two parts and then it gets subdivided until a single element remains. After that it will be sort according to the requirement (increasing or decreasing).  Now let’s see how an array is sorted. It requires three steps:   * Divide: Divide then n-element sequence to be sorted into two subsequence of n/2 elements each. * Conquer: Sort the two subsequences’ recursively using merge sort. * Combine: Merge the two sorted subsequence to produce the sorted answer.   Collectively it is known as **recursion**. So, in merge sort we call mergeSort function recursively. Now array is divided in such a way that we find the mid-point and divide that array into two sub arrays. Now left and right array is sorted individually. For left and right array, we again then divide from its mid-point and sort it. This process will go on till last element is left. Then it will be sorted accordingly. |
| PseudoCode | mergeSort (array, a, b)        If a == b              return        else              mid = (a + b)/ 2              mergeSort (array, a, mid)              mergeSort (array, mid+1, b)              merge (array, a, mid, b) |
| Python Code | def MergeSort(arr,a,b):     if (a == b):         return     else:         mid = (a+b)//2         MergeSort(arr,a,mid)         MergeSort(arr,mid+1,b)         Merge(arr,a,mid,b)  def Merge(arr,a,mid,b):     A = []     B = []     for i in range (a, mid+1):         A.append(arr[i])      for j in range (mid+1, b+1):         B.append(arr[j])          A.append(9999)     B.append(9999)     i = 0     j = 0      for k in range (a, b+1):         if(A[i] < B[j]):             arr[k] = A[i]             i = i + 1         else:             arr[k] = B[j]             j = j + 1 arr = [] n = int (input ("Enter number of elements: ")) for i in range (0, n):     num = int (input ())     arr.append(num) a = 0  b = len(arr)-1 MergeSort(arr,a,b) print ("Output: " + str(arr)) |
| Time Complexity | |  |  |  | | --- | --- | --- | | Worst Case | Average Case | Best Case | | 0 (n log n) | 0 (n log n) | 0 (n log n) | |
| Proof of Correctness | Loop invariant: At the start of each iteration of the for loop, the subarray A[j..n]A[j..n] consists of the elements originally in A[j..n]A[j..n] before entering the loop but possibly in a different order and the first element A[j]A[j] is the smallest among them.  **Initialization:**  Initially the subarray contains only the last element A[n]A[n], which is trivially the smallest element of the subarray.  **Maintenance:**  In every step we compare A[j]A[j] with A[j - 1]A[j−1] and make A[j - 1]A[j−1] the smallest among them. After the iteration, the length of the subarray increases by one and the first element is the smallest of the subarray.  **Termination:**  The loop terminates when j = ij=i. According to the statement of loop invariant, A[i]A[i] is the smallest among A[i..n]A[i..n] and A[i..n]A[i..n] consists of the elements originally in A[i..n]A[i..n] before entering the loop**.** |
| Strengths | 1. It is efficient for larger number of inputs. 2. It has a consistent running time. 3. Time complexity is Ω (n log n) in worst case. |
| Weaknesses | 1. It is not efficient for smaller number of inputs. 2. It requires extra space to store subarrays. 3. The algorithm does the whole process even if the array is sorted. |
| Dry Run |  |

|  |  |
| --- | --- |
| ***Bubble Sort*** |  |
| Description | Bubble Sort is a **comparison-based** sorting algorithm. In bubble sort, it compares two adjacent indexes and checks if it is in order or not. It then swaps two elements if they aren’t in order and this process continuous until the array is sorted. |
| PseudoCode | BubbleSort(A) for i = 1 to A.length - 1         for j = A.length – 1                if A [j] < A [j-1]                        swap A [j] with A [j-1] |
| Python Code | def BubbleSort(arr):     for i in range(len(arr)-1):         for j in range (0, len (arr)-i-1):             if (arr[j] > arr[j+1]):                 arr[j], arr[j + 1] = arr[j + 1], arr[j]  arr = [9,6,2,7,12,0] BubbleSort(arr) print("Output: " + str(arr)) |
| Time Complexity | |  |  |  | | --- | --- | --- | | Worst Case | Average Case | Best Case | | O (n2) | O (n2) | O (n) | |
| Proof of Correctness | **Loop invariant:**  At the start of each iteration of the for loop, the subarray A[j..n]A[j..n] consists of the elements originally in A[j..n]A[j..n] before entering the loop but possibly in a different order and the first element A[j]A[j] is the smallest among them.  **Initialization:**  Initially the subarray contains only the last element A[n]A[n], which is trivially the smallest element of the subarray.  **Maintenance:**  In every step we compare A[j]A[j] with A[j - 1]A[j−1] and make A[j - 1]A[j−1] the smallest among them. After the iteration, the length of the subarray increases by one and the first element is the smallest of the subarray.  **Termination:**  The loop terminates when j = ij=i. According to the statement of loop invariant, A[i]A[i] is the smallest among A[i..n]A[i..n] and A[i..n]A[i..n] consists of the elements originally in A[i..n]A[i..n] before entering the loop. |
| Strengths | 1. It is a stable sorting algorithm. 2. It does not contain extra memory space. 3. It performs greatly when array is almost sorted. |
| Weaknesses | 1. It is not efficient for larger sets of numbers. 2. It is very expensive as O(n2) in worst case and average case. 3. It requires a lot of time. |
| Dry Run |  |

|  |  |
| --- | --- |
| ***Selection Sort*** |  |
| Description | It is a **comparison-based** sorting algorithm. It first finds the smallest element in an array and place it in the beginning. Then it finds the smallest element from remaining array and then place it at second index. We will keep finding minimum elements and place it in array. This is how selection sort sorts an array. |
| PseudoCode | SelectionSort(array,n) for i = 1 to n – 1          min = i          for j = i+1 to n                  if array[j] < array[min]                          min = j                 if indexMin! = i                           swap array[min] and array[i] |
| Python Code | def SelectionSort(arr):     for i in range(0, len(arr)):         min = i         for j in range(i+1,len(arr)):             if(arr[j] < arr[min]):                 min = j         if(min != i):             key = arr[i]             arr[i] = arr[min]             arr[min] = key        arr = [6,9,2,3,0,12] SelectionSort(arr) print(arr) |
| Time Complexity | |  |  |  | | --- | --- | --- | | Worst Case | Average Case | Best Case | | O (n2) | O (n2) | O (n2) | |
| Proof of Correctness | First, we prove the correctness of the inner loop:  **Loop Invariant:**  Before the start of each loop, A[min] is less than or equal to A[i..j-1].  **Initialization:**  Prior to the first iteration of the loop, j=i+1. So the array segment A[i..j-1] is really just spot A[i]. Since min = i, we have that min indexes the smallest element (the only element) in subarray A [i…j-1] and hence the loop invariant is true.  **Maintenance:**  Before pass j, we assume that min indexes the smallest element in the subarray A[i..j-1]. During iteration j we have two cases: either A[j] < A[min] or A[j] ≥ A[min]. In the second case, the if statement is not true, so nothing is executed. But now min indexes the smallest element of A[i..j].  In the first case, switches min to index location j since it is the smallest. If min indexes an element less than or equal to subarray A[i..j-1] and now A[j] < A[min], then it must be the case that A[j] is less than or equal to elements in subarray A[i..j-1]. It switches min to index this new location and hence after the loop iteration finishes, min indexes the smallest element in subarray A[i..j].  **Termination:**  At termination of the inner loop, min indexes an element less than or equal to all elements in subarray A[i..n] since j = n+1 upon termination. This finds the smallest element in this subarray and is useful to us in the outer loop because we can move that next smallest item into the correct location**.** |
| Strengths | 1. It is efficient for smaller inputs. 2. It does not contain extra memory space. 3. It doesn’t depend on the initial arrangement of the elements. |
| Weaknesses | 1. It is not efficient for larger sets of numbers. 2. Its performance is easily influenced by the initial ordering of the items before the sorting process. 3. It requires n-squared number of steps for sorting n elements. |
| Dry Run |  |

|  |  |
| --- | --- |
| ***Quick Sort*** |  |
| Description | It is a **comparison-based** sorting algorithm. It chooses a pivot and place it in its position. All the elements greater than pivot are placed at right side and all the elements smaller than pivot are placed on left side. Then it executes recursively. |
| PseudoCode | quickSort(arr[], low, high)       if (low < high)             pi = partition(arr, low, high);             quickSort(arr, low, pi - 1)             quickSort(arr, pi + 1, high)  partition (arr[], low, high) pivot = arr[high] i = (low – 1) for (j = low; j <= high- 1; j++)          if (arr[j] < pivot)                 i++                 swap arr[i] and arr[j] swap arr[i + 1] and arr[high]) return (i + 1) |
| Python Code | def QuickSort(arr,low,high):     if(low < high):         index = partition(arr,low,high)         QuickSort(arr,low,index-1)         QuickSort(arr,index+1,high)          def partition(arr,low,high):     pivot = arr[high]     i = (low-1)     for j in range(low, high):         if(arr[j] < pivot):             i = i + 1             arr[i],arr[j] = arr[j],arr[i]     arr[i+1],arr[high] = arr[high],arr[i+1]     return (i+1)      arr = [] n = int(input("Enter number of elements: ")) for i in range(n):     num = int(input())     arr.append(num) low = 0 high = len(arr)-1 QuickSort(arr,low,high) print(arr) |
| Time Complexity | |  |  |  | | --- | --- | --- | | Worst Case | Average Case | Best Case | | O (n2) | O (nlogn) | O (nlogn) | |
| Proof of Correctness | The indices between j and r- 1 are not covered by any of the three cases, and the values in these entries have no particular relationship to the pivot x. We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.  **Initialization:**  Prior to the first iteration of the loop, i = p -1 and j = p. Because no values lie between p and i and no values lie between i +1 and j -1, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.  **Maintenance:**  Because of the swap, we now have that A[i <= x], and condition 1 is satisfied. Similarly, we also have that A[j-1 > x], since the item that was swapped into A[j-1] is, by the loop invariant, greater than x.  **Termination:**  At termination, j = r. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x. |
| Strengths | 1. It is efficient to deal with large number of input. 2. It requires only **n (log n)** time to sort **n** items. 3. No additional storage is required. |
| Weaknesses | 1. It is recursive function. 2. It requires O(n2) time in the worst case. 3. It is an unstable algorithm |
| Dry Run |  |

|  |  |
| --- | --- |
| ***Counting Sort*** |  |
| Description | This is a **linear-time** sorting algorithm. In counting sort, an array is sorted without comparison. It works with occurrences of each number in an array. Initially a count array is made which contains the number of occurrences of each element. Then that array is updated by adding two adjacent indexes. After that it is sorted in reverse order to maintain stability of the algorithm. It picks the last index of original array, goes to that index of count sort, decrement the value placed there and find that index in a new array and then place the value of original array at the index got by count array. This is ow counting sort works. |
| PseudoCode | function CountingSort(input)  k = range of elements of array  count ← array of k + 1 zeros  output ← array of same length as input   for i = 0 to length(input) - 1 do  j = key(input[i])  count[j] += 1  for i = 1 to k do  count[i] += count[i - 1]  for i = length(input) - 1 down to 0 do  j = key(input[i])  count[j] -= 1  output[count[j]] = input[i]  return output |
| Python Code | def countSort(arr):     size = len(arr)     arr2 = [0]\*size     maximum = max(arr)+1     count = [0]\*maximum          for j in range(size):         count[arr[j]] += 1        for k in range(1, maximum):         count[k] += count[k-1]          for m in reversed(arr):         count[m] -= 1         arr2[count[m]] = m              for n in range(size):         arr[n] = arr2[n]              return arr  arr = [] n = int(input("Enter number of elements: ")) for i in range(n):     num = int(input())     arr.append(num)  countSort(arr) print("Sorted elements are: " + str(arr)) |
| Time Complexity | |  |  |  | | --- | --- | --- | | Worst Case | Average Case | Best Case | | O (n+k) | O (n+k) | O (n+k) | |
| Proof of Correctness | **Initialization:**  The for loop of initializes the array count to all zeros, then the for loop inspects each input element. If the value of an input element is i, we increment C[i]. Thus, C[i] holds the number of input elements equal to i for each integer i =0,1,2 ... k. It determines for each i =0,1,2 .... k how many input elements are less than or equal to i by keeping a running sum of the array C**.**  **Maintenance and Termination:**  Finally, the for loop of lines 10–12 places each element A[j] into its correct sorted position in the output array B. If all n elements are distinct, then when we first enter line 10, for each A[j] , the value C[A[j]] is the correct final position of A[j] in the output array, since there are C[A[j]] elements less than or equal to A[j] . Because the elements might not be distinct, we decrement C[A[j]] each time we place a value A[j] into the B array. Decrementing C[A[j]] causes the next input element with a value equal to A[j] , if one exists, to go to the position immediately before A[j] in the output array. |
| Strengths | 1. **Linear time**: Counting sort runs in O(n) time. 2. It is a stable algorithm. 3. It is a good sorting algorithm if the range of input values isn’t greater than number of values to be sorted. |
| Weaknesses | 1. Counting sort can only sort distinct values. 2. It is not suitable for sorting large data sets 3. It is not suitable for sorting string values. |
| Dry Run |  |

|  |  |
| --- | --- |
| ***Radix Sort*** |  |
| Description | Radix sort is a non-comparative, **linear-time** sorting algorithm. It is similar to the radix sort except one thing, which is that it sorts number according to the positions. For example, a number 238 contains three digits, so radix sort will first sort the number according to *tens* position which is 8. Then by *hundred* position which is 3, and at the end by *thousand* position which is 2. Likewise counting array, it first creates a count array and place the number of occurrences (first sorting with tens position) in it. Then it updates that array by adding adjacent indexes. At the last it will start in reversed order i.e., picks 8th value, goes to the 8th index of count array, decrements it value, and places the value placed at index 8th of original array at index found by count array (the decremented value). It then does the same process for 3 and 2. This is first pass. This is how all the values gets sorted. |
| PseudoCode | def radixSort(A):        position = 1     digit = Floor of (maximum/position)      while(digit > 0) do         Call countSort with arguments A, position         position \*= 10         digit = Floor of (maximum/position)     end while     return A  def countSort(A, position)     Initialize count array of size 10 with 0.     Initialize array2 of A size with 0.          for i=1 to size do         count[(Floor of A[i]/position)mod(10)] += 1     end for      for j =2 to len(count) do         count[j] = count[j] + count[j-1]        end for          for k=A.length downto 1          var = (Floor of k/position)mod(10)         count[var] = count[var] - 1         array2[count[var]] = k     end for          for l =1 to size do         A[l] = array2[l]         end for |
| Python Code | def radixSort(arr):        position = 1     digit = maximum//position      while(digit > 0):         countSort(arr, position)         position \*= 10         digit = maximum//position              return arr  def countSort(arr, position):     count = [0] \* 10     arr2 = [0] \* size          for i in range(size):         count[((arr[i]//position)%10)] += 1              for j in range(1, len(count)):         count[j] += count[j-1]             for k in reversed(arr):         count[((k//position)%10)] -= 1         arr2[count[((k//position)%10)]] = k          for l in range(size):         arr[l] = arr2[l]  arr = [] n = int(input("Enter number of elements: ")) for i in range(n):     num = int(input())     arr.append(num)      size = len(arr) maximum = max(arr) radixSort(arr) print("Sorted array is: "+ str (arr)) |
| Time Complexity | |  |  |  | | --- | --- | --- | | Worst Case | Average Case | Best Case | | O (nk) | O (nk) | O (nk) | |
| Proof of Correctness | This algorithm works because it is stable: amongst keys with equal value, their relative orders are preserved. The formal proof of correctness applies the following loop invariant.  **Loop invariant:**  In the outer for loop, just before the iteration with a particular value of i, the integers in L are sorted according to the values induced by their last i digits, di-1, ... , d2, d1, d0  **Proof (by induction).**  [Basis] This statement implies that before the iteration with i=0, they are not sorted at all.  This is trivially true. 7  Induction step: Assume that just before theiteration with a particular value of i, the integers in L are sorted according to the integers induced by their last i digits. We want to prove that after the iteration with i, the values in L are sorted according to the integers induced by their last i+1 digits,di, di-1, ... , d2, d1, d0. |
| Strengths | 1. It is a stable sorting algorithm. 2. Unlike counting sort, it can easily deal negative numbers. 3. When there are smaller number of elements, radix sort works fast. |
| Weaknesses | 1. Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. 2. Radix sort is slower than other sorting algorithms. 3. It takes more space. |
| Dry Run |  |

|  |  |
| --- | --- |
| ***Bucket Sort*** |  |
| Description | Bucket Sort is a **linear-time** sorting algorithm. In this algorithm we make buckets according to the size of array. If the size of array is 10, we create 10 buckets. Now let’s see how bucket sort works. Take a value 0.562. As the first digit of this value is 5, so bucket sort will place this value at bucket 5. It then places all values to the respective buckets. If there are two values in the form 0.562 and 0.563. It places both these values at bucket 5 and then we will apply any sorting algorithm to sort these values or we call also call bucket sort recursively. This process will be applied for all the values present in an array. |
| PseudoCode | bucketSort(arr[], n) 1) Create n empty buckets (Or lists). 2) Do following for every array element arr[i]. .......a) Insert arr[i] into bucket[n\*array[i]] 3) Sort individual buckets using insertion sort. 4) Concatenate all sorted buckets. |
| Python Code | import math def bucketSort(arr,size):     arr2 = []     for i in range(size):         arr2.append([])     for j in range(len(arr)):         index = math.floor(arr[j] \* size)          arr2[index].append(arr[j])          for k in range(size):         arr2[k] = insertionSort(arr2[k])              m = 0     for i in range(len(arr)):         for j in range(len(arr2[i])):             arr[m] = arr2[i][j]             m += 1     return arr  def insertionSort(array):     for j in range(1, len(array)):         key = array[j]             i = j - 1         while i >= 0 and array[i] > key:             array[i+1] = array[i]             i = i - 1         array[i+1] = key     return array  arr = [0.897, 0.565, 0.665, 0.1234, 0.656, 0.3434, 0.414, 0.414, 0.230, 0.14, 0.54] size = len(arr) bucketSort(arr,size) print("Sorted array is: " + str(arr)) |
| Time Complexity | |  |  |  | | --- | --- | --- | | Worst Case | Average Case | Best Case | | O (n2) | O (n+k) | O (n+k) | |
| Proof of Correctness | **Initialization, Maintenance and Termination:**  To see that this algorithm works, consider two elements A[i]and A[j]. Assume without loss of generality that A[i] A[j] . Since bnA[i]<=A[j] , either element A[i] goes into the same bucket as A[j] or it goes into a bucket with a lower index. If A[i] and A[j] go into the same bucket, then the for loop puts them into the proper order. If A[i] and A[j] go into different buckets, puts them into the proper order. Therefore, bucket sort works correctly. |
| Strengths | 1. It is an efficient algorithm as each bucket is proceed independently. 2. It is good with floating point numbers. 3. Bucket sort works best if data is evenly distributed. |
| Weaknesses | 1. If the buckets are distributed incorrectly, you may wind up spending a lot of extra effort for no or very little gain. 2. It is not good with numbers equal to or greater than 1. 3. It can’t be applied to all data types. |
| Dry Run |  |

## **Sorting Filters:**

**Alphabetically**: It sorts the data alphabetically, in either ascending order or descending order.

**Price range**: It sorts the data in either ascending order of price or descending order of price.

## **Searching Filters:**

**Starts with**: The searching algorithm will search data according to the first letter. It will display all the products starting with that “specific” letter.

**Ends with**: The searching algorithm will search data according to the last letter. It will display all the products ending with that “specific” letter.

**Contains with:** The searching algorithm will search data according to the letter mentioned here. It will display all the products containing that “specific” letter.

## **Multi-Level Sorting:**

In multi-level sorting, one column is sort according to any other column. Let’s say we have sort books alphabetically (A-Z), now the next column is of price. So, the first book which starts from A must have the lowest price in all of the books.

# Collaboration

|  |  |
| --- | --- |
| Time Management | The time was managed well. We divided the work and done it according to the suitable time of each person. But there is one thing that there was a communication gap. So, we just discussed that who will work which task and get it done. |
| Problems Faced | **Scraping:**  Scrapping was a difficult task. It was basically a new thing. So we started it with scrapping a single entity, then a single page and then the complete website. Plus, it was difficult to find website because each website has different attributes. Besides this at first when 2020-CS-150 was scrapping the data, suddenly one million entities came to csv file but it was duplicate products. So, she has then again done scrapping from start and it almost took a day.  **Integration:**  It was the difficult task as pyqt5 was not working and then there were errors of “File not Found.” And it was quite difficult to rearrange data into the table. |
| Leadership | We have led in the task that was divided. 2020-CS-124 has done reports and 2020-CS-150 have done implementation. |

# Task Division

|  |  |  |
| --- | --- | --- |
|  | 2020-CS-124 | 2020-CS-150 |
| Mid Term Proposal | I have done pencil work. And give the basic proposal template. | I written the proposal report. And improved the pencil work. |
| GUI | I have designed the GUI. | I have improved the GUI. |
| Sorting Algorithms | I have given 4 sorting algorithms. Besides this I have given proof of correctness of all sorting algorithms. | The sorting algorithms given by 2020-CS-124 has some issues like indentation and in preview. So, I have re-written the algorithms again. I have given dry run and code for all sorting algorithms along with the pseudo code and description. |
| Scraping | She did some searching and scrap some data using the same code. | I have done all the scrapping and scrapped almost 24,000 data. |
| Integration | Didn’t do. | I have updated the GUI created by 2020-CS-124 and load the data from csv file into GUI table widget. |
| Final-Mid Report | I have filled the collaboration table and task division table. | I have written the document and updated the work done by 2020-CS-124. |

**Github Repository Link:**

<https://github.com/HafsaRashid/CS261F21PID35>