**Data Structure and Algorithms.**

**Midterm Project Final Report.**



**Project Title:**

Hotel Pricing Analytics

**Submitted To:**

Sir Samyan Qayyum Wahla

**Submitted By:**

Aysha Aysha Shabbir

(2020-CS-67) (2020-CS-66)

**Section:**

B

**Submission Date:**

03-11-2021

**Department of Computer Science.**

**UNIVERSITY OF ENGINEERING AND TECHNOLOGY, LAHORE.**

Table of Contents

**Aims and Objectives3**

Learning Outcomes3

Data Scrapping4

Details4

Source4

Attribute Details4

Project UI5

UI5

Component Details6

Algorithms5

Sorting Algorithms6

Searching Algorithms21

Integration23

Team Work24

**Aims and Objectives**

Aims and objectives of this project are:

* To make a project that named as hotel pricing analytics. As the name indicate, this user friendly system will collect a large amount of data about hotels and display it to user who required this information.
* User Interface will be simple and user friendly.
* This project is especially helpful and useful for those who often go for hoteling for any reason whether for official or entertainment trip to other countries and they want information such as charges hotels in different countries of world.
* This system will sort data by applying different sorting algorithms such as counting sort, bucket sort, insertion sort, selection sort, quick sort etc.
* This system will also search data by applying different searching algorithms such as binary search and linear search.
* This project will work on data obtained by web scrapping. By which large amount of data is scrapped from a website named as Booking.com. Different sorting and searching algorithms will be applied on this data.

**Learning Outcomes**

In this section, we will throw light on learning outcomes that is what we learnt from this project names as hotel pricing analytics.

Following are the learning outcomes:

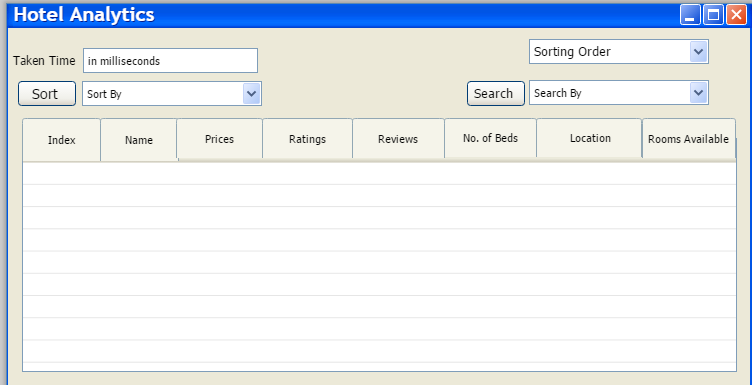
* Clear concepts of Data Structure
* Implementation of different sorting algorithms on large group of data
* Implementation of different searching algorithms on large group of data
* Implementation of multi-level sorting
* Implementation of composite filters
* Web Scrapping
* Python language
* UI using pyQt
* Integration of UI
* Designing a user friendly UI

**Data Scrapping**

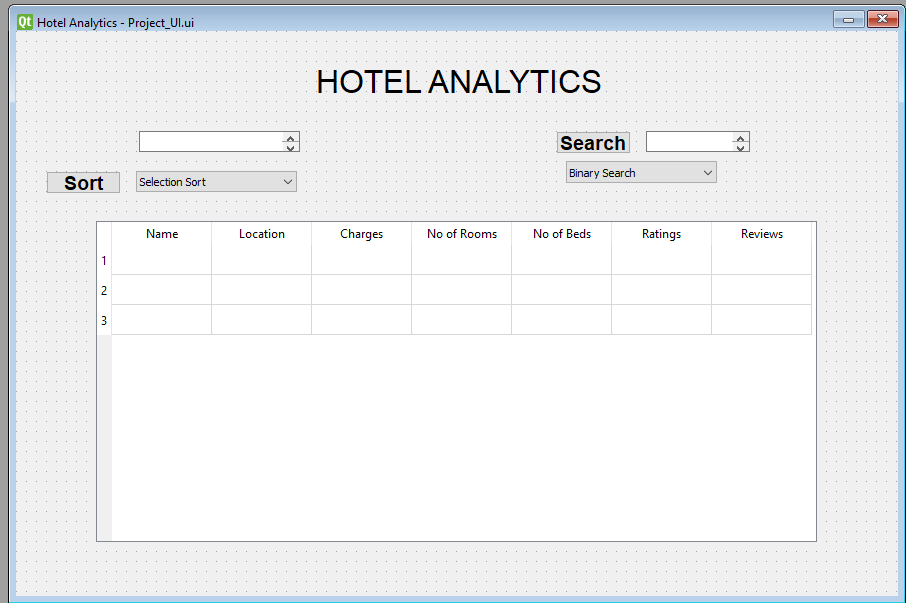
|  |  |
| --- | --- |
| **Details** | Data Scrapping is the first and most important step of this project.  Data scrapping is the best way to get a large amount of data from websites having huge amount of data.  In order to implement different sorting and searching algorithms, we require data. So in this project we scrapped a large group of data from Booking.com. This website contains data of worldwide hotels country and region wise. From this website we scrapped data about worldwide hotels, because this project is about hotels analytics. We gathered data about hotels in order to apply different algorithms on it as per requirements of this project.  Different types of sorting and searching algorithms are applied on this scrapped data having different attributes. |
| **Source** | Source of scrapping is:  <https://www.booking.com/index.en-gb.html?aid=100994;sid=35e8f7e4419f617e5860e97b5ccf72df;sig=v12tGK1KnU> |
| **Attribute Details** | |  |  |  | | --- | --- | --- | | Name | Data Type | Description | | Name of Hotel | String | This attribute is for storing name of hotel. | | Location | String | This attribute is for storing location of hotel. | | Prices | Integers | This attribute will store prices of hotels. | | Ratings | String | This attribute will store ratings of hotels. | | Reviews | String | This attribute will store reviews about hotels. | | No of rooms | Integers | This attribute will store no of rooms available in the hotel. | |

**Project UI**

* Pencil tool design is:



* pyQt UI is:



**Component Details:**

|  |  |  |
| --- | --- | --- |
| UI Component Name | Type of UI component | Purpose of UI Component/Other details |
| Data Showing Table | Table | Purpose of this component is to show data according to requirement of user. |
| Label for Time | Label | Purpose of this component is to label the time in milliseconds. |
| TextField for Time | TextField | Purpose of this text field is to show time taken in milliseconds. |
| Sorting order dropdown | Dropdown | Purpose of this component is to show  different options of sorting whether  ascending/descending or alphabetic  order to user. |
| Searching dropdown | Dropdown | Purpose of this component is to show  different options of searching algorithms  which to select. |
| Sorting algorithms dropdown | Dropdown | Purpose of this component is to show  different options of sorting algorithms  which to select. |
| Label for Sorting | Label | Purpose of this component is to label the  sotings. |
| Label for Search | Label | Purpose of this component is to label the  searching. |
| Button to load Data | Button | Function of this button is to load data. |

**Algorithms**

In this project, when large amount of data will be collected by scrapping, then different type of algorithms will be applied on it in order to perform operations.

Following algorithms will be applied on scrapped data:

* Sorting algorithms
* Searching algorithms
* Multi-level Sorting
* Composite filters

**Sorting Algorithms:**

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

|  |  |
| --- | --- |
| **Insertion Sort** |  |
| Description | Insertion sorting is a sorting algorithm in which items are transferred one by one to the correct position. In other words, an insert sort creates the final sorted list, one item at a time, as the higher-ranked items move. Sorting by insertion has the advantages of simplicity and a low additional cost .In an insert sort, the first element of the array is considered sorted, even if it is an unsorted array on each iteration, the sort algorithm removes one element at a time and finds the appropriate position in the sorted array and places it there. Iteration continues until the entire list is sorted. Insertion sorting has many advantages. It is simple to implement and efficient enough for small datasets, especially if they are mostly sorted. It has low overhead and can sort the list as it receives data. Another benefit associated with insert sorting is that it only needs a constant amount of memory space for the entire operation. It is more efficient than other similar algorithms such as bubble sort or select sort. |
| Pseudo Code | Insertion Sort(Array)  1. for j=2 to Array.length  2. key= A[j]  3. i = j-1  4. while i>0 and A[i] > key  5. A[i+1] = A[i]  6. i=i-1  7. A[i+1] = key |
| Python Code | def InsertionSort(A):  s= len(A)  for i in range(1, s): key = A[i]  j = i-1  while j >=0 and key < A[j] : A[j+1] = A[j]  j =j- 1  A[j+1] = key |
| Time Complexity Analysis | **Total Running Time**  (T(n)) = C1 \* n + (C2 + C3) \* (n - 1) + C4 \* Σ n - 1j = 1(t j) + (C5 + C6) \*  Σ n - 1j = 1(t j) + C8 \* (n - 1)  **Running Time for Worst case** (Array is fully un-sorted)  **T (n) = O (n2)**  **Running Time for Good case** (Array is nearly sorted) For Best Case i.e., tj **= 1**  After Substitution,  T (n) = C1 \* n + (C2 + C3) \* (n - 1) + C4 \* (n - 1) + (C5 + C6) \* (n - 2) + C8 \* (n - 1)  **T (n) = O (n)** |
| Proof of Correctness | Correctness can be proved through three steps:  **1.Initialization**  In fact, the loop invariant is always valid for the first element before the actual iteration of the loop begins. As can be seen in the pseudo-code, before the first iteration j = 2 At this point the sorted array would only have one element A [1] and if an array has one element it is assumed to be sorted. Therefore, the loop invariant is checked before the first iteration of the loop.  **2.Maintenance**  Now I will try to show that each element in the array keeps the loop invariant. As stated before, the body of the for loop works to move the element to the right and insert the desired element in the appropriate place, e.g. A [j1], A [j2] and so on. The sub-array is therefore made up of all the sorted elements. We then increment j to preserve and maintain the flow of the for loop for the next iteration. Thus, in this external and internal manner, the loops are fully maintained*.*  **3.Termination**  Finally, let's take a look at what happens when the circle ends. The condition causing the for loop to end is that j> A. length  = n. Since each circular loop constructs j by 1, we should then have j= n + 1. By substituting n + 1 for j in the invariant phrasing of the circle, we have that the sub-array A [1 ... n] includes the components initially in A [1 ... n] but all at once upon agreed request. Since the sub-table A [1 ... n]. |
| Strengths | * Good Performance for lesser element. * It requires minimal space. * It is simple. |
| Weaknesses | * Bad Performance for greater elements. * Useful for only lesser elements. * Requires a large number of shifting. |
| Dry Run | Input:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 33 | 10 | 26 | 11 | 30 |   Sorted Part:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 33 | 10 | 26 | 11 | 30 |   After i=1   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 33 | 26 | 11 | 30 |   After i=2   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 26 | 33 | 11 | 30 |   After i=3   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 11 | 26 | 33 | 30 |   After i=4   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 11 | 26 | 30 | 33 | |

|  |  |
| --- | --- |
| **Selection Sort** |  |
| Description | The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two sub arrays in a given array:  1) The sub array which is already sorted.  2) Remaining sub array which is unsorted  The sort-by-selection algorithm sorts an array by repeatedly finding the smallest element (considering ascending order) of the unsorted part and inserting it at the beginning. The algorithm maintains two sub-arrays in a given array  It plays thoroughly on small lists. It is an in-vicinity algorithm. It does now no longer require a number of area for sorting. Only one more area is needed for containing the temporal variable. It plays nicely on objects which have already been sorted. |
| Pseudo Code | SelectionSort(A, s)   1. N=length.A 2. For j in N 3. min = j 4. for k = j+1 to N 5. If ( A[i] < A[min] 6. Min = 1 7. Swap ( A [i] , A [ min ] ) |
| Python Code | def selectionSort(arr, n):  for i in range(n):  min=i  for i in range(i+1,n):  if arr[i]<arr[min]:  min=i  (arr[i],arr[min])=(arr[min],arr[i]) |
| Time Complexity Analysis | **Total Running Time**  (T(n)) = C1 \* 1 + (C2 + C3) \* (n - 1) + C4 \* Σ n - 1j = 1(n-j+1) + (C5 + C6) \* Σ n - 1j = 1(n-j+1) + C7\* (n-1)  **Running Time for Worst case** (Array is fully un-sorted)  **T (n) = O (n2)**  **Running Time for Good case** (Array is nearly sorted)  **T (n) = O (n)** |
| Proof of Correctness | Correctness can be proved through three steps:  **1.Initialization**  Selection sort start loop over each index of the array. Prior to the first iteration of the loop, for the outer loop we what we do is select and element from the array and consider it to be the smallest element from the entire array. And if we assume the entire array to be consisting of a single element that it a=is already sorted and for the first index, array is wholly sorted.  **2.Maintenance**  Now what we will do is to prove that loop variation is get maintained for each iteration of the loop. In the first call of min, it has to look for every element present in the array and so the body of the loop will run n times. N here represents the number of elements in the entire array. Now first elements get sorted and now loop will iterate from 1 to n index of the array and thus making swapping if appropriate. For the further iteration loop will iterate and will go through n-1 to n elements of the array.  **3.Termination**  At the point of termination, loop iteration get equal to n. Now, the array contains smallest elements of after comparing it with minimum which is thus the sorted array. It’ll get terminated when the whole array is sorted by going though each iteration of the loop invariant and thus algorithm is proved to correct. |
| Strengths | * Good performance on lesser elements * No temporary storage is required * Simplicity |
| Weaknesses | * Bad performance for greater elements * Worst Time Complexity is entirely bad |
| Dry Run | Input:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 33 | 10 | 26 | 11 | 30 |   Sorted Part:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 33 | 10 | 26 | 11 | 30 |   After i=1   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 33 | 26 | 11 | 30 |   After i=2   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 26 | 33 | 11 | 30 |   After i=3   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 11 | 26 | 33 | 30 |   After i=4   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 11 | 26 | 30 | 33 | |

|  |  |
| --- | --- |
| **Bubble Sort** |  |
| Description | Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping adjacent items if they are in the wrong order  Bubble sorting is mainly used for educational purposes to help students understand the basics of sorting. It is used to identify if the list is already sorted. When the list is already sorted (which is the best case), the complexity of the bubble sort is only Greater)  Instead of finding an array as a whole, bubble sort works by comparing pairs of adjacent objects in the array. If the items are not in the correct order, they are swapped so that the larger of the two moves upward. ... The exchange continues until the whole table is in the correct order. |
| Pseudo Code | Bubble Sort (A)   1. For i=0 to len(A)-1 2. For j=0 to len(A) – i -1 3. If A[j] > A[j+1] 4. Swap A[j] and A[j+1] |
| Python Code | def BubbleSort(arr):  n=len(arr)  for i in range(0,n):  for j in range(0,n-1):  if (arr[j]<arr[j+1]):  temp=arr[j]  arr[j]=arr[j+1]  arr[j+1]=temp  return arr |
| Time Complexity Analysis | **Running Time for Worst case** (Array is fully un-sorted)  T (n) = O (n2)  **Running Time for Good case** (Array is nearly sorted) For Best Case i.e., j **= 1**  After Substitution,  T (n) = C1 \* n + (C2 + C3) \* (n - 1) + C4 \* (n - 1)  T (n) = O (1) |
| Proof of Correctness | Correctness can be proved through three steps:  **1.Initialization**  Before the first iteration of the loop, when a single after has not been called, we can see from the pseudo code that sizei1 which means that if the length of an array is 8, then the last element d 'an array that would be index 7 would be the smallest element in the array. In general, A [n1] would be the smallest element in an array.  **2.Maintenance**  Now, we have to show how our algorithm maintains the loop invariants and prove that if this is true for k, it should also satisfy all the steps k + 1. At each step of the loop, we compare A [j + 1] and A [j] and check if A [j + 1] is less than A [j] if this becomes true, then A [j] and A [j +1] are swapped. In this whole scenario, after each iteration, the length of the sorted array starts to increase, and after the first iteration, the first element is the smallest element in the sub array  **3.Termination**  The loop will end when i of the outer loop becomes equal to j of the inner loop. According to the variant, A [i] is the minimum element of the sub-array and is originally composed of the elements of A [i ... n] before the iteration of the loop. |
| Strengths | * Simplicity * No temporary memory is required * Perform at constant time if array is sorted |
| Weaknesses | * It is inefficient for greater elements * Lack of efficiency * Takes greater time to sort elements |
| Dry Run | Input:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 33 | 10 | 26 | 11 | 30 |   After i=0   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 33 | 10 | 26 | 11 | 30 |   After i=1   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 33 | 26 | 11 | 30 |   After i=2   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 26 | 33 | 11 | 30 |   After i=3   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 11 | 26 | 33 | 30 |   After i=4   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 11 | 26 | 30 | 33 | |

|  |  |
| --- | --- |
| **Counting Sort** |  |
| Description | Count sorting is a sorting technique based on keys within a specific range. It works by counting the number of objects with distinct key values ​​(hash type). Then do some calculations to calculate the position of each object in the output sequence. Sorting by number is an efficient algorithm for sorting an array of elements each of which has a non-negative integer key, for example an array, sometimes called a list, of positive integers may have keys that are just the value of l 'integer as a key, or a wordlist can have keys assigned by some pattern. The major advantage of counting sort is its complexity , where is the size of the sorted array and is the size of the assistant array (range of distinct values). |
| Pseudo Code | CountingSort(A)  1.for i = 0 to k do  2.c[i] = 0  3.for j = 0 to n do  4.c[A[j]] = c[A[j]] + 1  5.for i = 1 to k do  6.c[i] = c[i] + c[i-1]  7.for j = n-1 downto 0 do  8.B[ c[A[j]]-1 ] = A[j]  9.c[A[j]] = c[A[j]] - 1  10.end func |
| Python Code | def Count\_Sort(A):  n=len(A)  C=[0]\*10  Res=[0]\*n  for i in range(0,n):  C[A[i]]=C[A[i]]+1  for i in range(1,10):  C[i]=C[i]+C[i-1]  i=n-1 |
| Time Complexity Analysis | **Running Time for Worst case**  T(n)= O(n^2)  **Running Time for Good case**  T(n)=O ( n ) O(n) O(n) |
| Strengths | * Understandable * Simplicity * Lesser time complexity |
| Weaknesses | * Only used for the array with integers * Array of frequencies cannot be constructed * Lack of efficiency for integer arrays |
| Dry Run | Input:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 2 | 1 | 1 | 4 | 2 |   Making Count Array:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 0 | 0 | 0 | 0 | 0 |   After calculating frequencies:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 0 | 2 | 2 | 0 | 1 |   Cumulative Sum:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 0 | 2 | 4 | 4 | 5 |   After iterations:  Output:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 1 | 1 | 2 | 2 | 4 | |

|  |  |
| --- | --- |
| **Radix Sort** |  |
| Description | Sort by base is an integer sorting algorithm that sorts data with integer keys by grouping keys by unique digits that share the same position and significant value (positional value). Sort by base uses sort by number as a routine to sort an array of numbers. Sorting by base is one of the sorting algorithms used to sort a list of integers in order. In the base sort algorithm, a list of integers will be sorted by the digits of the individual numbers .For example, if the largest number is a 3-digit number, the list is sorted in 3 steps. Sort by base is a non-comparative sorting algorithm that sorts items digit by digit from the least significant digit to the most significant digit. Suppose we want to sort 10 items in ascending order using sort by base, sort the unit location digit first. tenth digit. |
| Pseudo Code | Radix-Sort(A,d)  for j=1 to d  int count[10] = {0};  for i = 0 to n  count[key of(A[i]) in pass j]++  for k = 1 to 10  count[k] = count[k] + count[k-1]  for i = n-1 downto 0  result[ count[key of(A[i])] ] = A[j]  count[key of(A[i])]--  for i=0 to n  A[i] = result[i]  end for(j)  end func |
| Python Code | def countSort(array,loc):  n = len(array)  res = [0]\*n  count = [0]\*10  for i in range(0,n):  index = array[i]//loc  count[index % 10] += 1  for i in range(1,10):  count[i] += count[i - 1]  i =n-1  while i >= 0:  index = array[i]//loc  res[count[index % 10] - 1] = array[i]  count[index % 10] -= 1  i -= 1  for i in range(0,n):  array[i]=res[i]  #Function of radix sort  def radixSort(array):  Max=max(array)  loc = 1  while Max // loc > 0:  countSort(array,loc)  loc=loc\*10 |
| Time Complexity Analysis | **Running Time for Worst case**  T(n)= O(nk)  **Running Time for Good case**  T= O ( n ) O(n) O(n) |
| Strengths | * Fast when the keys are short * It is able to deal well with a huge list of items * No additional storage is required as well |
| Weaknesses | * It is less flexible than other sorts * For arrays with integer elements * Array of frequencies cannot be constructed. |
| Input/Output | Input:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | | 125 | 400 | 577 | 32 | 5 | 55 |   Output:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | | 5 | 32 | 55 | 125 | 400 | 577 | |

|  |  |
| --- | --- |
| **Merge Sort** |  |
| Description | The merge sort calculation intently follows the divide and conquer approach. In division, divide the sequence of n components to be organized into two sequels of n = 2 components each. By conquer the two sequels recursively using the consolidated sort.  Approach:  The critical task of calculating the consolidation sort is the convergence of two arrangements arranged in the “join” step. We converge by calling a supporting methodology MERGE (A, p, q, r) where A is an array and p, q is rare indices. MERGE sort reserves O (n) time. Merge sort is one of the most efficient sorting techniques. |
| Pseudo Code | Merge(A, p, q, r)   1. n1= q-p+1 2. n2= r-q 3. L1[n1] 4. R[n2] 5. For i=1 to n1 6. L[i] = A[p+i-1] 7. For j=1 to n2 8. R[j] = A[q + j] 9. i = j = 1 10. For k = p to r 11. If L[i] < M[j] 12. A[k] = L[i] 13. i = i+1 14. Else: 15. A[k] = R[j] 16. j= j+1 17. While( i < and = q) 18. A[k] = L [i] 19. i=i+1 , k= k+1   Merge Sort(A, p ,r)   1. If p < r 2. Q = (p + r)/2 3. Merge Sort(A ,p ,q) 4. Merge Sort(A ,q+1 ,r)   Merge( A, p , q, r ) |
| Python Code | def mergeSort(Arr):  m=len(Arr)  if len(Arr)>1:  r=int(len(Arr)/2)  Left=Arr[:r]  Right=Arr[r:]  mergeSort(Left)  mergeSort(Right)  i=0  j=0  k=0  while (i<len(Left) and j<len(Right)):  if Left[i]<0:  Arr[k]=Left[i]  i=i+1  k=k+1  else:  Arr[k]=Right[j]  j=j+1  k=k+1  while i<len(Left):  Arr[k]=Left[i]  i=i+1  k=k+1  while j<len(Right):  Arr[k]=Right[j]  j=j+1  k=k+1 |
| Time Complexity Analysis | **Total Running Time**  (T(n)) = O(n)  **Running Time for Worst case**  T (n) = O (n log m)  **Running Time for Good case and Average case**  T (n) = O (n log n) |
| Proof of Correctness | Correctness can be proved through three steps:  **1.Initialization**  The critical task of calculating the consolidation sort is the convergence of two arrangements arranged in the “join” step. We converge by calling a supporting methodology MERGE (A, p, q, r) where A is an array and p, q is rare indices. MERGE sort reserves O (n) time.  **2.Maintenance**  Now we need to see that our algorithm maintains cycle variations, to justify it we assume that L [i] <and> R [i] so the respective lines will perform precise and useful loop invariants.  **3.Termination**  At the endpoint, k is equal to r plus 1. Now the sub array contains the smallest elements of L [i] and R [i] which is then the sorted array. It will be finished when the whole array is sorted going through each iteration of the loop invariant, then it is shown that the algorithm corrects. |
| Strengths | * Good performance for greater elements * It is a stable sort and has consistent running time * It breaks the entire array into two parts and lead towards the solution parallel. |
| Weaknesses | * Not commendable for lesser elements * Flows through the entire elements of array * Worst Space Complexity |
| Dry Run | Input:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | | 45 | 40 | 57 | 22 | 5 | 25 |   Output:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | | 5 | 22 | 25 | 40 | 45 | 57 | |
| **Quick Sort** |  |
| Description | Quick Sort is a Divide and Conquer algorithm.It takes an element as apivot and partitions the given array around the selected pivot.There are many different versions of quick sort which select the pivot in different ways.  1. Always choose the first element as the pivot**.**  2. Always choose last element as pivot (implemented below)  3. Choose random element as pivot.  4. Choose the median as the pivot.  The key process in quick sort is partition **().** The purpose of partitions is, given an array and an array element x as a pivot, to put x in its correct position in the sorted array and put all elements smaller **(**smaller than x) before x, and put all major elements (greater than x) after x.All of this has to be done in linear time. |
| Pseudo Code | If low<high  Pi=P(A,low,high)  Quicksort(a,low,Pi-1)  Quicksort(a,Pi+1,high) |
| Python Code | def get\_pivot(list,first,last):  pivot=list[first]  left=first+1  right=last  while True:  while left<=right and list[left]<=pivot:  left=left+1  while left<=right and list[right]>=pivot:  right=right-1  if right<left:  break  else:  list[left],list[right]=list[right],list[left]  list[first],list[right]=list[right],list[first]  return right  def quick\_sort(list,first,last):  if first<last:  p=get\_pivot(list,first,last)  quick\_sort(list,first,p-1)  quick\_sort(list,p+1,last) |
| Time Complexity Analysis | **Running Time for Worst case**  T(n)= O(n^2)  **Running Time for Good case**  T(n)= n\*log(n) |
| Strengths | * Understandable * It has very small inner loop * n (log n) time to sort n items |
| Weaknesses | * It is recursive * It takes a quadratic time (i.e**.** n2) in the worstcase * It is fragile |
| Input/Output | Input:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | | 125 | 400 | 577 | 32 | 5 | 55 |   Output:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | | 5 | 32 | 55 | 125 | 400 | 577 | |

**Searching Algorithms:**

* Linear Search
* Binary Search

|  |  |
| --- | --- |
| **Linear Search** |  |
| Description | Linear search is simplest way to search an element in an array. It is also very basic searching algorithm. This algorithm works in such way that it sequentially moves through data, searching for required element. In simple words, it looks through the list one item at a time not by jumping.  Now let’s talk about that how this algorithm work. First of all, it start from the leftmost element of the array and one by one compare the searching element with each element of the array. If any element of array matches with the element we are searching for, then return index of that element and stop searching further. Whereas if searching element is not present in the array then return -1. |
| Pseudo Code | linearSearch(arr, value)  1. for each item in the list  2. if match item == value  3. return the item's location  4. end if  5.end for  end func |
| Python Code | def linearSearch(arr,size,x):  for i in range(0,size):  if (arr[i]==x):  return i  return -1 |
| Time Complexity Analysis | Time complexity is:  O(n) |
| Strengths | * Fast **searching of small to medium lists.** * **Array don’t need to start.** * **No effects of insertions and deletions.** |
| Weaknesses | * It is not useful for big arrays. * Searching for large lists is slow. |
| Dry Run | Item to Search: 20  Input:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 12 | 18 | 20 | 25 |   Output:  Searched Item index=3. |

|  |  |
| --- | --- |
| **Binary Search** |  |
| Description | Binary Search is a fast searching algorithm. This algorithm works on the principle of divide and conquer. A condition is that data collection must be in the sorted form for this algorithm to work properly.  This algorithm searches a particular element by comparing the middle most item of the array. If the searching item is found, then its index is returned. If the middle item is greater than the element, then the item is searched in the sub array to the left of the middle item. Else the element is searched for in the sub array to the right of the middle item. This is continues on the sub array until the size of the sub array reduces to zero. |
| Pseudo Code | binarySearch(arr,item,start,last)  if start<=last  midIndex = (start+ladt)/2  if item == arr[midIndex]  return midIndex  else if item < arr[midIndex]  return binarySearch(arr, item, midIndex + 1,last)  else  return binarySearch(arr, item,start,midIndex - 1)  return -1 |
| Python Code | def binarySearch(arr, item, start, last):  if last >= start:  midIndex = start + (last - start)//2  if arr[midIndex] == item:  return midIndex  elif arr[midIndex] < item:  return binarySearch(arr, item, start, midIndex-1)  else:  return binarySearch(arr, item, midIndex + 1, last)  else:  return -1 |
| Time Complexity Analysis | * Best Time Complexity: O(1) * Average Time Complexity: O(logn) * Worst Time Complexity: O(logn) |
| Strengths | * Better time complexity * Breaks array into sub arrays * More efficient for large data |
| Weaknesses | * Can be implemented over a sorted array only. * Has poor locality of reference. |
| Dry Run | Item to Search: 20  Input:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 10 | 12 | 18 | 20 | 25 |   start =0, last=4, mid=2   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | **2** | 3 | 4 | | 10 | 12 | 18 | 20 | 25 |   start =3, last=4, mid=3   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | **3** | 4 | | 10 | 12 | 18 | **20** | 25 |   Hence index of searched item=3 |

**Integration**

Integration is the process by which we combine UI with functions of the programs. UI of this project will be designed by using pyQt software. Code of this project will be in python language. We will implement different sorting and searching algorithms by integration on this UI.

Integration is:

Scrapped data+UI+Algorithms(python code)

* Functions of sorting algorithms will be implemented on the scrapped data.
* These sorting algorithms include insertion, merge, quick, counting, radix, selection and bubble sort.
* Different searching functions will also be implemented over this scrapped data.
* These searching algorithms include binary search and linear search.
* Multi-level sorting will also be implemented.
* Different composite filters will also be implemented over scrapped data.
* All these functions will be implemented over scrapped data by integration using UI.

**Team Work**

**Group Members:**

|  |  |
| --- | --- |
| **Name** | **Registration Number** |
| Aysha | 2020-CS-67 |
| Ayesha Shabbir | 2020-CS-66 |

* Both the members worked on milestones of this project.
* First of all, we selected the project title and then make project proposal.
* Then we start work on scrapping.
* After that start working on project UI.
* And then finally work on integration and final report.
* We did this project by ms teams meeting as well as physical meetings on campus.