Central Real Estate Scrapper



Project Supervisor

Mr. Samyan Qayyum Wahla

Project Members (Group 31)

|  |  |
| --- | --- |
| Haider Ali | 2020-CS-137 |
| Ali Tariq | 2020-CS-142 |

Department of Computer Science

University of Engineering and Technology, Lahore

Pakistan

Table of Contents

[Project Description: 3](#_Toc71624246)

[Motivation: 4](#_Toc71624247)

[Business Need: 4](#_Toc71624247)

[Level of Impact: 4](file:///C:\Users\Ali%20tariq\Desktop\Project%20Requirement%20Dcoument%20CS162.docx#_Toc71624247)

[Entities: 4](file:///C:\Users\Ali%20tariq\Desktop\Project%20Requirement%20Dcoument%20CS162.docx#_Toc71624247)

[Technology Stack: 5](#_Toc71624248)

[Scrapping Details: 6](#_Toc71624249)

[User Interface Design: 4](#_Toc71624247)

[Algorithms: 4](file:///C:\Users\Ali%20tariq\Desktop\Project%20Requirement%20Dcoument%20CS162.docx#_Toc71624247)

Sorting [Algorithms: 7](#_Toc71624250)

[Insertion Sort: 7](#_Toc71624251)

[Merge Sort: 7](#_Toc71624251)

[Selection Sort: 7](#_Toc71624251)

[Insertion Sort: 7](#_Toc71624251)

[Quick Sort: 7](#_Toc71624251)

[Radix Sort: 7](#_Toc71624251)

[Bucket Sort: 7](#_Toc71624251)

[Counting Sort: 7](#_Toc71624251)

[Heap Sort: 7](#_Toc71624251)

[Shell Sort: 7](#_Toc71624251)

[Hybrid Sort: 7](#_Toc71624251)

[Tree Sort: 7](#_Toc71624251)

Searching [Algorithms: 7](#_Toc71624250)

[Binary Search: 7](#_Toc71624251)

[Linear Search: 7](#_Toc71624251)

[Collaborations/Task Division: 11](#_Toc71624260)

[Project Plan 15](#_Toc71624264)

# Project Description:

Real estate advisors are facing the problems to keep record to all the properties, so the solution is done with the help of web scraping. So, a software have been designed in which the data will be scraped from different top class and authentic websites and in order to make it more convenient for the user to interact with it, there will also be a feature of sorting in which various sorting algorithms will be used to sort the data motivation for doing this is that the advisors will see the maximum or minimum range in which they have plots in accept of price the user also capable of sorting on the bases of no of rooms or the area or location. For example, users will be able to sort the data with the filters such as with entities like location, title, area and also with the price. This feature really help advisors because every customer is not of the same level each customer have its own demands and ranges so with sorted data the advisors will clearly tell them the range in which they can purchase or their needs will be fulfill or not.

Moreover, to make it more efficient, a proper graphical user interface (GUI) have been designed in which the user will have the following features:

* Scrape the data from the given website
* Search within data with data filters.
* Sort the data according to his own sorting method
* User will be given choice of different algorithms
* User will interact the application with proper graphical user interface (Designed with pyQT5)

Our system is capable of scrap data from some related website and not only scrapping but to make scrapping efficient we add a feature to pause and resume and cancel the scrapping and the progress of scrapping is shown in progress bar. When scrapping is done the data is loaded into a “.CSV” file where the data is stored permanently so data will be loaded into tables from CSV files if the user want fresh data he also have feature to live scarp the data and the enhanced features like pausing, resuming and stopping data scrapping are also available for the consumer.

We allow different type of sorting like ascending and descending and the user also select the sorting algorithm of his own choice (insertion sort, bubble sort, merge sort, quick sort) etc the option is given to change the algorithm in the UI as different algorithms takes different time so the choice of algorithm selection is of users and he will also test the difference between the working efficiency of different algorithms

We also provide different types of searching like from title of plot price area or the exact location and also with the price of the desired plot. We also provide some searching filters like search with (price, no of beds, no of baths, area, and location e.t.c.) so user will be able to search on the bases of that filters.

# Project Motivation:

# Our motivation is to help the estate advisors community who face a big problem to manage all the data of plots a city or an area as in this society everything is going to be computerized without computer we can’t compete with outer world so our motivation is to make Pakistan advance and compete with the top countries in technology fields. Our second motivation is that as we are student of computer sciences we need to explore things and in our this project we are learning an interesting technique which is called data scrapping this is new for us and seems to be quite interesting as we are going to collect data from a genuine website with our code and we have nothing to do apart from doing code and rest of data is scrapped so fully motivated for this new thing and motivated for exploring UI of python (pyQT5).

# Business Need:

As today there is a big need of property everyone needs his own house and the state advisors need to take as many as they can record of properties in order to earn more but they are un able to keep recode of every new plot that is available to sell so this system will help them as it scrap latest data of plots and the not need to physically move and search plots so they just our system and by sitting in front of their computer they can see more than one million properties and also compete in market.

As today market depends on latest things the advisors are still working with paper based system so whenever new plots came on quite some time they are un aware on it and between this time many of plots are sold so to overcome this problem they need a system to take recode of the latest items so this system will take latest information from authenticated websites and display it. Another big problem our advisors are facing is that every customers have its own range and desire so to search for desired plots for the customer from a lot of plots on papers is not so easy they might got frustrate so to overcome this problem we introduce sorting and searching in our system so the advisor sort the data to find desired range plots and then search for an appropriate plot for customer.

# End User:

The end users of the designed product are

* Estate advisors
* Property companies
* Person who wants to buy property

# Level of Impact:

There will be a huge impact and it will be for both the normal user and, but it will be a great beneficial for the real estate advisor, he will be able to know more about the properties in his area and it can cause a great profit in his revenue. On the other hand, the normal user can view in which areas the property is god to buy.

# Entities:

* Title
* Description
* Location
* Area
* Price
* Number of Bedrooms
* Number of Bathrooms

Above mentioned entities are in our system there details are:

|  |  |  |
| --- | --- | --- |
| **Name** | **Data Type** | **Description** |
| Title | String | The title of the property will show that with which title the person has named his property.  This is set up by seller and it have the little intro to property like where it located and how much it worth’s etc. |
| Description | String | A brief description about the property like some additional information the buyer must know is provided in here. The additional information includes the exact location area and the price also the number of beds and baths. This will help buyer to make a map in his mind about property. |
| Location | String | The location will indicate the location of the property, where it is located across the globe. |
| Area | String | The Area of the property will represent that how much area (It could be in any form like Marla, Kanal) the property has covered. |
| Price | Integer | Price of the property is displayed it might be in rupees or in dollars depends on the country. |
| No of bedrooms | Integer | It indicates the number of beds in the house if the house is not constructed then it will automatically store zero. |
| No of bathrooms | Integer | It indicates the number of bathroom in the house if the house is not constructed then it will automatically store zero.. |

# Technology Stack:

|  |  |
| --- | --- |
| Languages | Python |
| Platform | Desktop Application |
| Frontend Technology | Python Qt5 |
| IDEs | Jupiter, VS Code |
| UI Prototype | Pencil Tool |

# Algorithms:

* Sorting Algorithms

|  |  |
| --- | --- |
| **Insertion Sort** |  |
| Description | Insertion sort is one of the simplest type of sorting as sorting is done by iterating through the array using a key value every time a single key is selected and after iterate through the whole array the exact location of that particular key is found and hence one element is sorted in the same manner the whole the array is sorted in ‘n2’ iterations where n is the number of elements in array this sorting technique is simple but not suitable for large input as loop will be large and algorithm takes more time hence this algorithm is suitable for a limited input its implementation is simple we just take a key value usually 2nd element of array and iterate it through the array and compare it with all the index either it is larger or small then the neighbor index after finding its location the left side array is sorted and after all the loop iterations all the array is sorted although its a simple algorithm but it becomes so time consuming in its worst case as n2 time for n inputs if n becomes so larger the time of compilation becomes so large so this algorithm is only good for its best case but as new algorithms are found it is assumed that it can’t performs better than the advanced algorithm so it becomes useless as the same task is performed in less time. |
| Pseudo code | for j=2 to A.length( )  key=A[j]  i=j-1  while i>0 AND A[i]>key  A[i+1]=A[i]  i=i-1  A[i+1]=key |
| Python code | def insertion\_sort(arr):  for j in range(1, len(arr)):  key = arr[j]  i = j-1  while i >= 0 and key < arr[i] :  arr[i + 1] = arr[i]  i -= 1  arr[i + 1] = key |
| Time Complexity Analysis | C1: n  C2: (n-1)  C3: (n-1)  C4: Σn - 1j = 1(tj)  C5: Σn - 1j = 1 (tj - 1)  C6: Σn - 1j = 1(tj - 1)  C7: (n – 1)  Best Case: O(n)  Worst Case: O(n2)  Average Case: O(n2) |
| Proof of correctness | * **initialization:**   The first sub array is of one element and one element is assumed to be sorted   * **Maintenance :**   The iteration of loop increase the size of sub array and when new element enters the array it will maintain the sorting property of sub array it is inserted where it is larger than its left one element   * **Termination:**   The loop will be terminated when the sub array which is sorted to the size of actual array so the i indicated the size of original array. |
| Strengths | * Perform well for small no of inputs (best case) * Implementation is simple * Less space is required |
| Weakness | * As input is large it become time effective as n2 time is required * Not as efficient as other advanced algorithms * Need to iterate to all the array again and again |

|  |  |
| --- | --- |
| **Merge Sort** |  |
| Description | Merge sort is one of the most respected algorithm used in data structures it is an algorithm which uses divide and conquer rule to sort an array this sorting in worst case uses O(n log n) time the working technique is that it simply divide an array into 2 parts and keeps on doing so until array contains only one element as one element is already sorted so the merge sort algorithm keeps on dividing the array into 2 parts recursively until array size becomes one then the merge sort use a merge function which combines (conquer) all the one sized array into single array that the resulted array is fully sorted hence we say that merge sort only divide the arras and the merge function actually combines them in an sorted array the biggest factor about merge sort is that in best case, in worst case and in average case it takes the same time which is O(n log n) as this time is much more good than O(n2) so that is the one reason to use merge sort rather then insertion sort as it doesn’t depends on number of inputs it have same effect either we use it for large input or for small as recursion use O(n log n) so in case we have small input it becomes little costly as due to divide and conquer (recursion) some space is required on each call so it take more space than insertion sort as a result it is better to use merge sort algorithm for sorting large data. |
| Pseudo code | Merge\_Sort(A,a,b)  if(a==b)  return  else  m=floor(a+b/2)  Merge\_Sort(A,a,m)  Merge\_Sort(A,m+1,b)  Merge\_Sort(A,b,m)  return  merg(A,p,q,r)  n1=q-p+1  n2=r-q  L = [0] \* (n1)  R = [0] \* (n2)  for i=0 to n1  L[i] = A[p + i]  for j=0 to n2  R[j] = A[q + 1 + j]  i=0  j=0  k=p  while i < n1 and j < n2    if(L[i]<=R[j])  A[k] = L[i]  i=i+1  else  A[k] = R[j]  j=j+1  k=k+1  while i < n1  A[k] = L[i]  i += 1  k += 1  while j < n2  A[k] = R[j]  j += 1  k += 1 |
| Python code | def merg(A,p,q,r):  n1=q-p+1  n2=r-q  L = [0] \* (n1)  R = [0] \* (n2)  for i in range (0,n1):  L[i] = A[p + i]  for j in range(0,n2):  R[j] = A[q + 1 + j]  i=0  j=0  k=p  while i < n1 and j < n2:  if(L[i]<=R[j]):  A[k] = L[i]  i=i+1  else:  A[k] = R[j]  j=j+1  k=k+1  while i < n1:  A[k] = L[i]  i += 1  k += 1  while j < n2:  A[k] = R[j]  j += 1  k += 1  def mergeSort(A, p, r):  if((p<r)):  q=(p+r)//2  mergeSort(A,p,q)  mergeSort(A,q+1,r)  merg(A,p,q,r) |
| Time Complexity Analysis | Best Case: O(n log n)  Worst Case: O(n log n)  Average Case: O(n log n) |
| Proof of correctness | * **Initialization:**   At start the k has value p and sub array is empty as k-p=0 so the smallest elopements’ of L and R are considered as smallest elements. A contains a single component which is inconsequentially assumed to be sorted   * **Maintenance :**   If L[i]<=R[j] where L[i] is smallest element and not placed back to array so the increment in k and our own increment in i just to iterate through the right in sub array maintains the loop   * **Termination:**   When k=r +1 the sub array have smallest elements in left and right arrays and the sub array is now sorted so after iterating to all the sub array the loop terminated |
| Strengths | * Take less time for large data as compared to other algorithms * It is a stable sort * Use divide and conquer technique rather than iteration * It is consistent in time not depends on input |
| Weakness | * Not efficient for small input as compared for other algorithms * Requires more space for temporary arrays * In best case ( array is sorted ) it goes to same process takes O(n log n) time |

|  |  |
| --- | --- |
| **Selection Sort** |  |
| Description | It is one of the simplest type of sorting algorithm which only based on iteration on loop in selection sort we just find the minimum element of the array and place it at the start of the array as if we are sorting in lower to higher format then the smallest element is placed at the first index of the array in traversing one time in array and finding the minimum of it we almost need n time so after finding the smallest element and placing it on first index our array becomes small and now we find minimum from remaining elements of the array and place them according to their appearance so basically what we are doing is we just dividing our array in two parts left side is filled with sorted array and the right side is with unsorted and un checked elements after placing all the elements we see that our array is sorted the smallest element is selected from the array and replaced with the leftmost number in that order the array is sorted either its a simple sorting algorithm but this is not sufficient for large input of data because its time complexity is O(n2) so on large number of input it doesn’t perform well so it is suitable for small number of inputs. |
| Pseudo code | for i = 1 to n - 1  min = i  for j = i+1 to n  if list[j] < list[min] then  min = j;  if indexMin != i then  swap list[min] and list[i] |
| Python code | def selection\_sort(A):  arr=[]  for i in range(len(A)):  min\_idx = i  for j in range(i+1, len(A)):  if A[min\_idx] < 0:  min\_idx = j  temp=A[i]  A[i]=A[min\_idx]  A[min\_idx]=temp |
| Time Complexity Analysis | C1: 1  C2: (n-1)  C3: (n-1)  C4: Σn - 1j = 1(n-j+1)  C5: Σn - 1j = 1 (n-j)  C6: Σn - 1j = 1(n-j)  C7: (n – 1)  Best Case: O(n2)  Worst Case: O(n2)  Average Case: O(n2) |
| Proof of correctness | * **Initialization:**   At the start of the loop iteration the A[min index] is the smallest element of the array and i= min\_index   * **Maintenance:**   The both loops maintains the loop invariant as in outer loop we iterate through the array and the inner loop checks for min value of array when A[i]<A[min\_index ] then min\_index is updated   * **Termination:**   When the value of loop becomes equal to size of array then loop terminates after iterating and sorting all the elements in the array |
| Strengths | * Performance doesn’t affect by arrangement of data elements * Less operations are involved so where data movement is costly it is more economical * Simple to implement and understand |
| Weakness | * Less efficient as it takes O(n2) which is not good as compared to other algorithms * Worst case and bests case have no difference same time required * Not suitable for large data |

|  |  |
| --- | --- |
| **Bubble Sort** |  |
| Description | Bubble sort is the simplest type comparison based algorithm that use loop iterations to iterate through the array and find appropriate place for every number it uses two loops so its time complexity is O(n2) which means that if the bubble sort is applied on large input then it will not work properly now let us see how the bubble sort works it will take an array and with iteration of outer loop picks an element and entered in inner loop the in inner loop it iterates through the whole array and select the bests appropriate place for this element once the element is settled then the loop terminates and it enters in outer loop and now take the next element from the array and elaterid the inner loop here the appropriate place for this element is found when all the elements settle down to their place then array is sorted it is also a comparison based sorting now let’s talk about is performance on large or small input it has worst case time complexity O(n2) so on large input it is not suitable in best case ( array is sorted ) it takes O(n) time which is its little benefit over selection and insertion sort. |
| Pseudo code | for i = 0 to loop-1 do:  swapped = false  for j = 0 to loop-1 do:  if list[j] > list[j+1] then  swap( list[j], list[j+1] )  swapped = true  if(not swapped) then  break |
| Python code | def bubbleSort(arr):      n = len(arr)      for i in range(n-1):          for j in range(0, n-i-1):                if arr[j] > arr[j + 1] :                  arr[j], arr[j + 1] = arr[j+1],arr[j] |
| Time Complexity Analysis | C1: 1  C2: (n-1)  C3: Σn - 1j = 1(n-i+1)  C4: Σn - 1j = 1 (n-i)  C5: Σn - 1j = 1(n-i)  Best Case: O(n)  Worst Case: O(n2)  Average Case: O(n2) |
| Proof of correctness | * **Initialization:**   Initially the A[n] is considered the smallest element   * **Maintenance:**   The inner loop iterates through the array repeatedly and checks for the smallest elements with comparing with A[n] and n is decreasing then replace the element with its left size   * **Termination:**   The loop termination depends on the condition when loop variable becomes equal to size of array means loop iterated through all the array and now array is sorted |
| Strengths | * Take less time is bests case ( sorted input ) * Simple iterative strategy used * Less operations required data movement is not costly * Little memory overhead * Stable sorting algorithm |
| Weakness | * Takes more time as compared to other algorithms * Not suitable for large data * High number of swapping between elements |

|  |  |
| --- | --- |
| **Quick Sort** |  |
| Description | Just like merge sort the quick sort also uses the divide and conquer approach to sort an array the divide and conquer strategy basically designed to save the time of cpu hence the time required to sort an array is less in quick sort the array is also divided on the basis of a pivot element the pivot may be selected randomly or it might be selected according to some rule like select first element as pivot or either select the last one or the median of array the main part of quick sort is partition function this function after iterating through the array place the pivot x on its appropriate sorted location and after that it place all the elements small then pivot to left of that array and then the large elements to right of the array then quick sort recursively sort the array on the partitioned left and right arrays now talk about the time it have best case time O(n log n) and same for average case but for worst case( array is sorted ) it takes O(n2) time which is not suitable so it doesn’t performs well in its worst case. |
| Pseudo code | 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 partition(arr,low,high):  pivot=arr[high]  i=(low-1)  for j in range(low, high):  if arr[j] <= pivot:  i = i+1  temp=arr[i]  arr[i]=arr[j]  arr[j]=temp    temp=arr[i+1]  arr[i+1]=arr[high]  arr[high]=temp  return (i+1)  def quickSort(arr,low,high):  if(low<high):  pi=partition(arr, low, high)  quickSort(arr, low, pi - 1)  quickSort(arr, pi + 1, high) |
| Time Complexity Analysis | Best Case: O(n2)  Worst Case: O(n log n)  Average Case: O(n log n) |
| Proof of correctness | * **Initialization**:   Initially the pivot is set randomly and then the elements are compared with it   * **Maintenance**:   The loop is responsible for partition of the array the partition is done by comparing all the elements of loop with the pivot and the smallest element will be on left size of array he large elements on right of the array   * **Termination**:   The termination condition is simply when the low(first) becomes equal to high (last) index of the array |
| Strengths | * Use divide and conquer technique * Efficient for best and average case time is O(n log n) * It is able to deal well with a huge list of items. * No additional space required |
| Weakness | * Its worst case will perform same as insertion selection sort * If array is already sorted it is not suitable * If pivot is not selected well the time complexity will be affected |

|  |  |
| --- | --- |
| **Bucket Sort** |  |
| Description | Bucket sort is an interesting sorting algorithm in which an un sorted elements of array are placed into a bucket according to their weight the bucket is formed according to the range of the max number in the array then the number are placed in form of count in the respective bucket for e.g. if largest element in array id 60 then we form buckets of 60 size and place all the elements at the index equal to them at last all the sorted buckets are combined and a sorted array is formed its very simple but it have some disadvantages that if the largest element eve if one of them is very large then we need a large space to create the bucket also its time complexity is so varying that in normal case it is O(n) in best case it is O(n+k) and in worst case it becomes O(n2) the worst case is that all the elements in array are supposed to enter at same bucket it doesn’t depends on number of input rather it depends on the max of them it can’t applied on all data types it only work with integers |
| Pseudo code | bucketSort(array, k)  buckets = new array of k empty lists  M = max(array)  for i = 1 to length(array)    buckets[floor(k × array[i] / M)].append(array[i])  for i = 1 to k do  nextSort(buckets[i])  return buckets |
| Python code | def bucketSort(array):  bucket = []    for i in range(len(array)):  bucket.append([])  for j in array:  index\_b = int(10 \* j)  bucket[index\_b].append(j)  for i in range(len(array)):  bucket[i] = sorted(bucket[i])  k = 0  for i in range(len(array)):  for j in range(len(bucket[i])):  array[k] = bucket[i][j]  k += 1  return array |
| Time Complexity Analysis | Best Case: O(n+k)  Worst Case: O(n2)  Average Case: O(n) |
| Strengths | * As buckets are individual arrays so to sort them is more easy * Abel to use an external sorting algorithm * On best case it performs well also in average case |
| Weakness | * Need more spaces as the max in array increases the bucket sixe increases * In worst case it is not suit able as it take O(n2) time * Can’t work with all data types only integer * Performance depends on number of buckets |

|  |  |
| --- | --- |
| **Radix Sort** |  |
| Description | Radix sort is a sorting algorithm which sorts the elements in a linear time as it comes under the category which sorts in the linear time. Radix sort uses counting sort as base and then sort the element. In radix sort, elements are sorted digit by digit from least significant to most significant digit. After every iteration, next significant digit is sorted keeping in view the order which starts from tens then hundreds and so on. |
| Pseudo code | Radix\_Sort(A,d)  1- for i = 1 to d  2- use a stable sort to sort array A on digit i |
| Python Code | def CountSort(A):      max = int (FindMax(A))      length=len(A)        C= [0 for i in range (max+1)]      B= [0 for i in range (length)]        for j in range (0, len(A)):            C[A[j]] =C[A[j]] +1        for i in range (0, len(C)):          C[i]=C[i]+C[i-1]        i = length -1      while i >= 0:          C[A[i]]-=1          B[C[A[i]]-1] = A[i]          i -= 1      return B  def RadixSort(A):        maximum=FindMax(A)      Base=1      while maximum // Base>0:          B = CountSort(A)          Base \*=10      print(B) |
| Time Complexity Analysis | Best Case: O (n + k)  Worst Case: O (n + k)  Average Case: O (n + k) |
| Proof of correctness | * **Initialization**:   The correctness of radix sort follows by induction on the column being sorted Maintenance   * **Maintenance**:   The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to k, so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time. n C k.   * **Termination**:   There are d passes, and so the total time for radix sort is d\*n C k. When d is constant and k D O.(n), we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits. |
| Strengths | * When there are fewer elements, this radix sort is fast. * Radix sort is used in suffix array construction. * Radix Sort is also called stable sort. |
| Weakness | * Radix sort is not flexible; it needs to be re written for every different data type. * Extra space is required for this algorithm. |

|  |  |
| --- | --- |
| **Counting Sort** |  |
| Description | Counting sort is also a sorting algorithm that sorts an array of elements in the linear time. In this algorithm, an array is first analysed and the maximum element is found in an input array. A new array is made of the size of the maximum element in an input array. All the digits in the array are counted and placed in the temporary array after counting each of them. Now, the adjacent elements are added and stored on the index. At the last, we iterate through the input array and check all the digits one by one and stores the sorted array in a new array C. |
| Pseudo code | Counting\_Sort(A,B,k)  1 let C [ 0..k ] be a new array  2 for i = 0 to k  3 C=[i] = 0  4 for j = 1 to A.length  5 C[A[j]] = C [A[J]] + 1  6 // C[i] now contains the number of elements equal to i.  7 for i = 1 to k  8 C[i]= C[i] +C[i-1]  9 // C[i] now contains the number of elements less than or equal to i. 10 for j = A. length **down to** 1  11 B[C[A[j]]] = A[j]  12 C[A[j]] =C[A[j]]- 1 |
| Python Code | def CountSort(A):      max = int(FindMax(A))      length=len(A)        C=[0 for I in range (max+1)]      B=[0 for i in range (length)]      for j in range(0,len(A)):            C[A[j]]=C[A[j]]+1      for i in range(0,len(C)):          C[i]=C[i]+C[i-1]      i = length -1      while i >= 0:          C[A[i]]-=1          B[C[A[i]]-1]=A[i]          i -= 1      return B |
| Time Complexity Analysis | Best Case: O(n)  Worst Case: O(k)  Average Case: O(n+k) |
| Proof of correctness | * **Initialization**:   After the for loop of initializes the array C to all zeros, the for loop of lines 4–5 inspects each input element.   * **Maintenance**:   If the value of an input element is i, we increment C[i]. Thus, C[i]. holds the number of input elements equal to. |
| Strengths | * The running time of counting sort is O(n), when the length of the input array is not much smaller. * In counting sort, there is no comparison between elements. * Better for small range of elements. |
| Weakness | * It’s not good for large number of input. * Counting sort uses linear sort. * If input is very large, this algorithm does not work on it. |

|  |  |
| --- | --- |
| **Tree Sort** |  |
| Description | Tree sort is a sorting algorithm which uses some kind of tree like binary search tree to sort an input array. First of all, a binary search tree is made till the number of the inputs, after that the tree is traversed using in order traversal method on the tree and after performing all the above-mentioned steps, a sorted array is returned as output. |
| Pseudo code | Input array(A)  Make a binary search tree BST(A)  In order traversal(A) |
| Python Code | class Node:  def \_\_init\_\_(self, key):  self.key = key  self.left = None  self.right = None  self.parent = None    def insert(self, node):  if self.key > node.key:  if self.left is None:  self.left = node  node.parent = self  else:  self.left.insert(node)  elif self.key <= node.key:  if self.right is None:  self.right = node  node.parent = self  else:  self.right.insert(node)    def inorder(self):  if self.left is not None:  self.left.inorder()  print(self.key, end=' ')  if self.right is not None:  self.right.inorder()      class BSTree:  def \_\_init\_\_(self):  self.root = None    def inorder(self):  if self.root is not None:  self.root.inorder()    def add(self, key):  new\_node = BSTNode(key)  if self.root is None:  self.root = new\_node  else:  self.root.insert(new\_node)      bstree = BSTree()    alist = input('Enter the list of numbers: ').split()  alist = [int(x) for x in alist]  for x in alist:  bstree.add(x)  print('Sorted list: ', end='')  bstree.inorder() |
| Time complexity Analysis | Best Case: O(n log n)  Worst Case: O(n2)  Average Case: O(n) |
| Strengths | * Sorting in Tree sort algorithm is as fast as quick sort algorithm. * Like a linked list, in tree sort algorithm changes are vesy to make. |
| Weakness | * If the array is already sorted, worst case occurs. * In worst case, O(n^2) is the running time. |

# 

# Project Plan

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.#** | **Feature’s** | **Member Name** | **Estimated Completion Date** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |