

**Mid-Term Project Report**

**Project Name: TripAdvisor Scrapped**

**Project ID: 20**

**Group members:**

**2020-CS-23 Muhammad Rizwan**

**2020-CS-10 Asad Mehmood**

**Submitted to: Mr. Samyan Qayum Wahla**

**Department of computer science**

**University of Engineering and Technology, Lahore**

Contents

[Learning outcomes 3](#_Toc88207362)

[**1. Project overview** 3](#_Toc88207363)

[1.1. Description 3](#_Toc88207364)

[1.2. Motivation 3](#_Toc88207365)

[1.3. Targeted audience 3](#_Toc88207366)

[1.4. Business need 3](#_Toc88207367)

[**2. Project requirements** 4](#_Toc88207368)

[2.1. UI implementation 4](#_Toc88207369)

[2.2. Data 4](#_Toc88207370)

[2.3. Sorting Algorithms 4](#_Toc88207371)

[2.4. Searching algorithms 4](#_Toc88207372)

[2.5. Searching filters 4](#_Toc88207373)

[2.6. Multi-level sort 4](#_Toc88207374)

[**3. Project implementation details** 4](#_Toc88207375)

[3.1. Pre-project discussion 4](#_Toc88207376)

[3.2. Tool configuration 5](#_Toc88207377)

[**4. Sample of scrapping source** 5](#_Toc88207378)

[4.1. Data types 6](#_Toc88207379)

[**5. Details of sorting and searching algorithms** 7](#_Toc88207380)

[5.1. Sorting algorithms 7](#_Toc88207381)

[5.2. Searching algorithms 8](#_Toc88207382)

[**6. User Interface** 9](#_Toc88207383)

[**7. Project roadmap** 10](#_Toc88207384)

[7.1. Sorting algorithms Details 10](#_Toc88207385)

[7.1. Integration 33](#_Toc88207386)

[7.2. Final user interface 33](#_Toc88207387)

[7.3. Collaboration 34](#_Toc88207388)

# Learning outcomes

* Learn ways to scrap huge data from different websites.
* Learn how to sort huge amount of data using different sorting algorithms
* Learn skills to use PyQt designer
* Learn the use of appropriate data type as requirement

# **1. Project overview**

## 1.1. Description

TripAdvisor is a well-known platform for travelling advisory. They offer online hotel reservations and bookings for transportations, lodging, travel experiences and restaurants. The website has been in limelight for quite some time and has gained very much fame over the recent years. They operate worldwide and contain data of thousands of restaurants and hotels.

The data provided at the website is so diverse that the users sometimes feel non-confidant on what to choose. The data is so scattered that it will be difficult for the user to get the expected information easily. So, we will try to make their lives easy by providing them as simple information as possible. For this purpose, we will scrap the data into seven attributes. Each attribute will be providing data according to the necessity of the user. Multiple sorting algorithms will be provided so that the user can sort items in ascending or descending order. It will make it easy for user to get expected data. For example, the user can find the cheapest or most luxurious hotel in a matter of seconds. He can sort any column by using any of the given algorithms that will help him to get information from the required field.

When data is sorted, it is still so immense that it will take time to see the expected result. To compensate this problem, searching algorithms will also be provided so that the user can use any of the algorithms to search expected item from columns. We will provide multiple searching algorithms so that the maximum efficiency can be achieved. We will also provide an interface for stating, stopping and pausing the scrapping.

## 1.2. Motivation

The motivation of this project is to make lives of people easy. Many travellers find it difficult to search appropriate hotels in a new city or country. To make their lives simpler, we will provide them a complete list of hotels as per their requirement. This purpose will be achieved through multiple sorting and searching algorithms.

## 1.3. Targeted audience

The end user of this project will be travellers all around the globe as it will be helpful for them to find their desired hotel according to their needs.

## 1.4. Business need

The need for this project is that we will be helping people to find their desired hotel in the best way possible. They will be able to search their hotel according to their city and within their desired price range. The clients of the website won’t have to hustle through different pages searching for the right hotel. Instead, they will be able to jump to their result in a matter of seconds.

# **2. Project requirements**

## 2.1. UI implementation

* UI should be designed using PyQT
* One page should display the list of chosen entity

## 2.2. Data

* At least 1 million entities should be scrapped
* Scrapping tasks has the option to pause, start, resume and stop, with the progress bar showing the progress of tasks/ number of entities scrapped
* Attributes of entity should be at least 7

## 2.3. Sorting Algorithms

* UI should have the option for sorting of each column. User has choice to choose any algorithm for sorting for a particular column
* Number of algorithms for sorting should contain all those algorithms studied in class and at least three additional algorithms
* After sorting of column, display time in milliseconds

## 2.4. Searching algorithms

* UI should have the option for searching based on each column. User has choice to choose any algorithm for sorting for a particular column
* Advanced filters for string columns should be implemented such as contains, end with, starts with etc.

## 2.5. Searching filters

* User should have the option to search using composite filters such as AND, OR and NOT

## 2.6. Multi-level sort

* User should be option to sort using multiple columns

# **3. Project implementation details**

## 3.1. Pre-project discussion

Before the start of the project, we had to understand the requirements. Therefore, we had a few meetings in which we discussed about the website from which we had to scrap the data. We had to make sure that the website which we are choosing must have at least 7 attributes. After some web surfing, we finally landed on [www.tripadvisor.com](http://www.tripadvisor.com) . This website was a perfect match for our project as it contained all the 7 attributes which we were looking for.

After successfully deciding the website, we discussed about the amount of data which we were going to scrap. We expected to get at least half a million data. We also discussed which sorting and searching algorithms we were going to use during the project.

The next step was to design the wireframes of the project. For this, we used a pencil tool and made a simple UI design in it. We designed a proposal which was accepted, after which we divided the work among ourselves and started the project.

## 3.2. Tool configuration

The requirement of the project was to use PyQt5 for User interface. So, we used a tool called ‘PyQt Designer’. This tool was easy to use and allowed us to use the drag and drop feature. We designed UI almost as similar as the one shown in Pencil Tool.

**4. Sample of scrapping source**

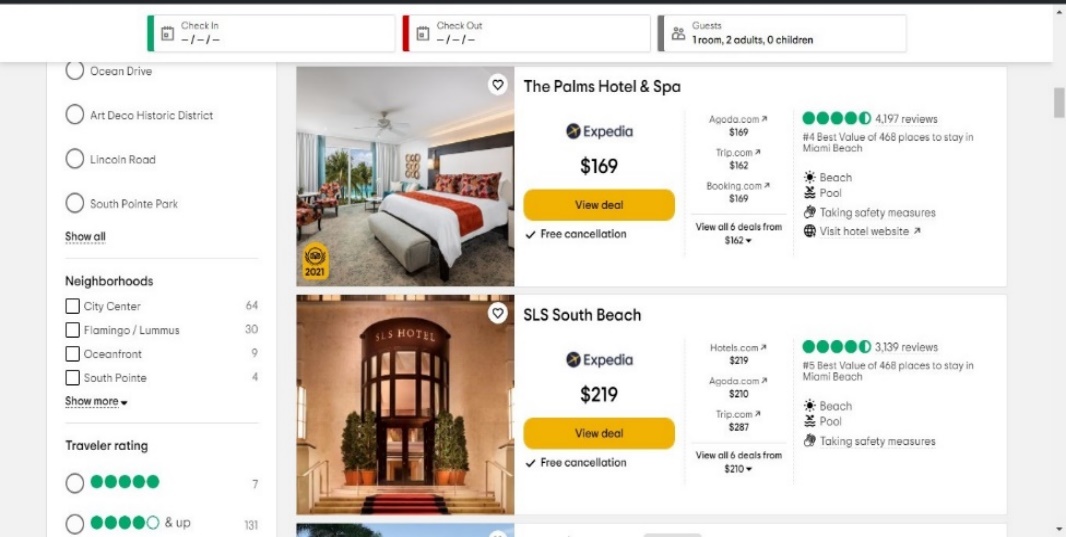


Figure 1: sample of source data for scrapping

## 4.1. Data types

|  |  |
| --- | --- |
| Name of Entity | Hotels |
| Attributes of Entity  (Minimum seven attributes/rows can be increased) | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Description** | | Name | String | This attribute will show the name of hotel. | | Price | Int | This attribute will show the price of stay at hotel. | | Rating | Int | This attribute will show the rating of hotel. | | Review | Int | This attribute will show number of hotels’ reviews. | | City | String | This attribute will show in which city the hotel is located. | | Services | String | This attribute will show which extra services will be provided at the hotel. | | Ranking | Int | This attribute will show the city-wise ranking of the hotel. | |

Table 1:Data types of scrapping data

# **5. Details of sorting and searching algorithms**

## 5.1. Sorting algorithms

Here is a brief description of every sorting algorithm used in this project.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Algorithm Name** | **Description** | | Bubble Sort | It is a type of sorting in which we repeatedly compare adjacent elements and swap them if they are in wrong order. The process is repeated until array is sorted. | | Insertion Sort | In this type of sorting, the first element in array is taken as sorted and second element is taken as ‘key’ and it is compared with every element in array to place them in their correct position. | | Selection Sort | In this type of sorting, we repeatedly search for smallest element in array and locate it in the beginning of array. | | Merge Sort | This type of sorting is based on divide and conquer rule. In this sorting, array is divided again and again until it remains with single element then it is sorted and combined recursively. | | Bucket Sort | This type of sorting distributes every element in buckets. Then every bucket is sorted by using insertion sort algorithm. | | Counting Sort | This sorting works by iterating through elements, counting the occurrence of every element present in array and use these counts to compute an element index in the final sorted array | | Radix Sort | Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value. Radix sort uses counting sort as a subroutine to sort an array of numbers. | | Quick Sort | Quick sort is just like Merge sort as it also uses divide and conquer rule. In this algorithm, we first select pivot and on the basis of pivot, we divide smaller and larger elements to their correct side. | | Shell Sort | It is type of sorting which first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted. The interval between the elements is reduced based on the sequence used. | | Cycle Sort | **Cycle sort** divides an array into the number of cycles. Then each cycle is rotated to produce a sorted array. | | Tim Sort | Tim sort first analyses the list it is trying to sort and then chooses an approach based on the analysis of the list. Tim sort actually makes use of Insertion sort and Merge sort. |   Table 2:Details of sorting algorithms |

## 5.2. Searching algorithms

Here is a brief description of the searching algorithms used in this project

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Searching Algorithms | |  |  | | --- | --- | | Linear search | A linear search is the simplest methodof searching a data set. We start at the beginning of the data set and examine each item of data until a match is made. | | Binary search | Binary search is used to search items in a list of sorted array. It keeps dividing array into half until the desired result is found. | | Jump search | With Jump Search, the sorted array of data is split into subsets of elements called blocks. We find the search key (input value) by comparing the search candidate in each block. As the array is sorted, the search candidate is the highest value of a block. | |
| Searching Filters for each data type | The searching filters will be as follows:   * Terms starting with \_\_\_\_\_\_ * Terms ending with \_\_\_\_\_\_ | |

Table 3: Details of searching algorithms and filters

# **6. User Interface**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **UI Component Name** | **Type of UI component** | **Purpose of UI Component** | | Name | Column | Show the names of hotels | | Price | Column | Shows the prices offered by hotels | | City | Column | Shows the city in which the hotel is located | | Ratings | Column | Shows the ratings of the hotel | | Reviews | Column | Shows the number of reviews gained by the hotel | | Rankings | Column | Shows city wise ranking of the hotel | | Start | Button | A button to start the scrapping | | Stop | Button | A button to stop the scrapping | | Pause | Button | A button to pause the scrapping | | Resume | Button | A button to resume the scrapping | | Progress bar | Bar | Shows the progress of the scrapping process | | Time elapsed | Label | Shows the time elapsed during the sorting | | Sorting algorithms | Drop-box | Shows the list of available sorting algorithms | | Searching algorithms | Drop-box | Shows the list of available searching algorithms | | Sort | Button | Sorts the selected column according to selected algorithm | | Search | Button | Searches the entire column with selected algorithm | | Ascend/descend | Button | Select the order of sorting. | | Terms to search | Textbox | Receives the information for the words to be searched | | Multi-sort | Checkbox | If selected, the column will be sorted according to the corresponding columns. |   Table 4:Details of buttons in User Interface (UI) |

# **7. Project roadmap**

* First of all, we completed our sorting algorithms with all details as mentioned below
* In the next step we implemented our UI (user interface) with the help of PyQt designer
* Then we implemented the code for scrapping the data from our pre-decided website: tripadvisor.com
* Then we arranged our scrapping code and integrated it with our UI code from PyQt designer.
* In the next step, we integrated our sorting and searching algorithms with UI design
* Then we put start, stop, pause, resume button to practical use
* After that we implemented the multi searching filters
* In the end we scrapped the data from website and presented the data into table, each attribute under the respective column

## 7.1. Sorting algorithms Details

|  |  |
| --- | --- |
| **Algorithm** | **Bubble sort** |
| Description | It is a type of sorting in which we repeatedly compare adjacent elements and swap them if they are in wrong order. The process is repeated until array is sorted. |
| Pseudo code | Let n = length of array  For i=0 to n-1 do  For j=0 to n-1 do  If array[j]>array[j+1] do  Swap array[j] and array[j+1] |
| Code | def **bubble\_sort**(array):      for i in **range**(**len**(array)-1):          for j in **range**(**len**(array) - 1):              if array[j] > array[j+1]:                  temp=array[j]                  array[j] = array[j+1]                  array[j+1]=temp    **print**(array) |
| Time complexity | for i in **range**(**len**(array)-1): n times    for j in **range**(**len**(array) - 1): (n)(n-1)       if array[j] > array[j+1]: (n)(n-2)           temp=array[j] (n)(n-2)           array[j] = array[j+1] (n)(n-2)           array[j+1]=temp (n)(n-2)  The best case has time complexity : O(n)  The average case has time complexity: O(n^2)  The worst case has time complexity :O(n^2) |
| Strengths | * Easy to understand * Very short code * Takes low memory |
| Weaknesses | * Takes longer time to sort * Not suitable for larger inputs |
|  |  |
| **Algorithm** | **Merge sort** |
| Description | This type of sorting is based on divide and conquer rule. In this sorting, array is divided again and again until it remains with single element then it is sorted and combined recursively. |
| Pseudo code | Merge-sort(A,p,r):  If p<r do  Q=p+r/2  Merge-sort(A,p,q)  Merge-sort(A,q+1,r)  Merge(A,p,q,r)  Merge(A,p,q,r):  num1=q-p+1  num2=r-q  let Left[1,…,num1+1] and Right[1,….,num2+1] be new arrays  for i=1 to num1 do  Left[i]=A[p+i-1]  for j=1 to num2 do  Right[j]=A[q+j]  Left[num1+1] = ∞  Right[num2+1] = ∞  i = 1  j = 1  for k = p to r:  if Left[i]<Right[j]:  A[k]=Left[i]  i = i+1  else A[k]= = Right[j]:  j = j+1 |
| Code | import **sys**  def **merge**(arr,p,q,m):      Left=[]      Right=[]      for i in **range**(p,m+1):          Left.**append**(arr[i])      for i in **range**(m+1,q+1):          Right.**append**(arr[i])      Left.**append**(**sys**.maxsize)      Right.**append**(**sys**.maxsize)      j=0      i=0      for idx in **range**(p,q+1):          if Left[i]<=Right[j]:              arr[idx]=Left[i]              i+=1          else:              arr[idx]=Right[j]              j+=1    def **merge\_sort**(arr,p,q):      if p<q:          mid=(p+q)/2          mid=**int**(mid)  **merge\_sort**(arr,p,mid)  **merge\_sort**(arr,mid+1,q)    **merge**(arr,p,q,mid) |
| Time complexity | import **sys**  def **merge**(arr,p,q,m):      Left=[] O(1)      Right=[] O(1)      for i in **range**(p,m+1): O(n)          Left.**append**(arr[i]) O(n-1)      for i in **range**(m+1,q+1): O(n)          Right.**append**(arr[i])       O(n-1)      Left.**append**(**sys**.maxsize) O(1)      Right.**append**(**sys**.maxsize) O(1)      j=0 O(1)      i=0 O(1)      for idx in **range**(p,q+1): O(n)          if Left[i]<=Right[j]: O(n-1)              arr[idx]=Left[i] O(n-1)              i+=1 O(n-1)          else: O(n-1)              arr[idx]=Right[j] O(n-1)              j+=1 O(n-1)    def **merge\_sort**(arr,p,q):      if p<q: O(1)          mid=(p+q)/2 O(1)          mid=**int**(mid) O(1)  **merge\_sort**(arr,p,mid)  **merge\_sort**(arr,mid+1,q)    **merge**(arr,p,q,mid)  The best case has time complexity : O (n.log n)  The average case has time complexity: O (n.log n)  The worst case has time complexity : O (n.log n) |
| Proof of correctness | **Initialization**: The loop invariant holds prior to the first iteration of the loop. Here, i = j = 1, and S is completely empty. L[1] is the smallest element of L, while R[1] is the smallest element of R, so the initialization step holds.  **Maintenance**: To see that each iteration maintains the loop invariant, suppose that L[i] ≤ R[j]. Then L[i] is the smallest element not yet copied to S. The current nonempty part of S consists of the k − 1 smallest elements, so after the loop is over and L[i] is copied to S, the nonempty part of S will consist of the k smallest elements.  Incrementing k (in the for-loop update) and i re-establishes the loop invariant for the next iteration.  **Termination:** At termination, k = m+ 1. By the loop invariant, S contains the m smallest elements of L and R, in sorted order. This is the result that we wanted (i.e., the merging of the two sorted arrays to produce a new sorted array) |
| Strengths | * Takes lesser time to sort * Constant time for any number of inputs * Works on divide and conquer technique |
| Weaknesses | * Takes same time even for smaller inputs * Not faster than quick sort * The code will run even if the array is already sorted |
|  |  |
| **Algorithm** | **Bucket sort** |
| Description | This type of sorting distributes every element in buckets. Then every bucket is sorted by using insertion sort algorithm. The final array is produced by putting out all the elements from bucket. |
| Pseudo code | Let arr be the array containing unsorted elements.  Let b\_array be a new array containing 10 more arrays.  For i in arr do  index = 10 \* j  index = int(index)  b\_array[idx] = j  for i = 1 to 10 do  b\_array[i]= insertion\_sort(b\_array[i])  for i = 1 to 10 do  for j= 0 to b\_array.Length do  array[k]= b\_array[i][j]  k++ |
| Code | def **bucketSort**(array):      b\_arrays=[]      for i in **range**(10):          b\_arrays.**append**([])      for j in array:          idx = **int**(10 \* j)          b\_arrays[idx].append(j)      for i in **range**(10):          b\_arrays[i] = **insert\_sort**(b\_arrays[i])      k=0      for i in **range**(10):          for j in **range**(**len**(b\_arrays[i])):              array[k] = b\_arrays[i][j]              k += 1  **print**(array)    def **insert\_sort**(array):      for i in **range**(1, **len**(array)):          key = array[i]          j = i – 1          while j >= 0 and array[j] > key:              array[j + 1] = array[j]              j=j-1          array[j+1] = key      return array |
| Time complexity | def **bucketSort**(array):      b\_arrays=[]      for i in **range**(10):          b\_arrays.**append**([])      for j in array:          idx = **int**(10 \* j)          b\_arrays[idx].append(j)      for i in **range**(10):          b\_arrays[i] = **insert\_sort**(b\_arrays[i])      k=0      for i in **range**(10):          for j in **range**(**len**(b\_arrays[i])):              array[k] = b\_arrays[i][j]              k += 1  **print**(array)    def **insert\_sort**(array):      for i in **range**(1, **len**(array)):          key = array[i]          j = i – 1          while j >= 0 and array[j] > key:              array[j + 1] = array[j]              j=j-1          array[j+1] = key      return array  The best case has time complexity : O (n+k)  The average case has time complexity: O (n)  The worst case has time complexity : O (n^2) |
| Strengths | * We sort buckets, which are smaller parts of the original array * Its independent of the comparison approach * It is a stable sort |
| Weaknesses | * Only works for decimal values * Consumes more memory due to empty buckets * Cannot be used for words and letters * Depends on insertion sort |
| **Algorithm** | **Cycle sort** |
| Description | **Cycle sort** divides an array into the number of cycles. Then each cycle is rotated to produce a sorted array. |
| Pseudo code | Cycle\_sort(arr):  For k= 0 to arr.Length do  temp=arr[k]  index=k  for i = k+1 to arr.Length do  if arr[i]<temp:  index = index + 1  while temp= =arr[index] do  index = index+1  if index != k do  swap temp with arr[index]  while index != k do  index= k  for i= start + 1 to arr.Length do  if arr[i] < temp do  index = index +1  while temp = = arr[index] do  index =index +1  swap temp with arr[index] |
| Code | def **cycleSort**(array):     cycle = 0     for k in **range**(0, **len**(array) - 1):          temp = array[k]          idx = k          for i in **range**(k + 1, **len**(array)):              if array[i] < temp:                  idx += 1          if idx == k:              continue          while temp == array[idx]:              idx += 1          temp2=array[idx]          array[idx]=temp          temp=temp2          cycle += 1          while idx != k:              idx = k              for i in **range**(k + 1, **len**(array)):                  if array[i] < temp:                      idx += 1              while temp == array[idx]:                  idx += 1              temp2=array[idx]              array[idx]=temp              temp=temp2              cycle += 1     return cycle |
| Time complexity | def **cycleSort**(array):     cycle = 0     for k in **range**(0, **len**(array) - 1): n times          temp = array[k] n-1 times          idx = k n-1 times          for i in **range**(k + 1, **len**(array)):              if array[i] < temp:                  idx += 1          if idx == k:              continue          while temp == array[idx]:              idx += 1          temp2=array[idx]          array[idx]=temp          temp=temp2          cycle += 1          while idx != k:              idx = k              for i in **range**(k + 1, **len**(array)):                  if array[i] < temp:                      idx += 1              while temp == array[idx]:                  idx += 1              temp2=array[idx]              array[idx]=temp              temp=temp2              cycle += 1     return cycle  The best case has time complexity : O (n^2)  The average case has time complexity: O (n^2)  The worst case has time complexity : O (n^2) |
| Strengths | * It is memory efficient * It is also storage efficient |
| Weaknesses | * The time complexity is always n^2 * It is unstable sort |
|  |  |
| **Algorithm** | **Radix sort** |
| Description | Radix sort is a sorting algorithm that sorts data with according to position of digits in numbers. First the unit places are used as reference to sort numbers, then tens place and so on. Radix sort uses counting sort at last to sort an array of numbers. |
| Pseudo code | Radix\_sort(arr):  Largest number=max(arr)  m=max number of elements in largest number  Create a new array of size of arr  For i= 0 to m do  Call the counting sort  Count\_sort(arr,numb):  Largest=max(array)  Let count\_array be a new array[0,….,largest+1]  For i= 0 to n: |
| Code | def **count\_sort**(array, const):      large = **max**(array)      n = **len**(array)      final = [0] \* (n)      count\_array = [0] \* (large+1)        for i in **range**(0, n):          idx =  array[i] // const          count\_array[idx % 10] += 1        for i in **range**(1, large+1):          count\_array[i] += count\_array[i - 1]      i = n - 1        while i >= 0:          idx = array[i] // const          final[count\_array[idx % 10] - 1] = array[i]          count\_array[idx % 10] -= 1          i -= 1      i = 0      for i in **range**(0, **len**(array)):          array[i] = final[i]    def **radixSort**(array):      const = 1      max\_value = **max**(array)      while max\_value / const > 0:  **count\_sort**(array, const)          const = const \* 10 |
| Time complexity | def **count\_sort**(array, const):      large = **max**(array)      n = **len**(array)      final = [0] \* (n)      count\_array = [0] \* (large+1)        for i in **range**(0, n):          idx =  array[i] // const          count\_array[idx % 10] += 1        for i in **range**(1, large+1):          count\_array[i] += count\_array[i - 1]      i = n - 1        while i >= 0:          idx = array[i] // const          final[count\_array[idx % 10] - 1] = array[i]          count\_array[idx % 10] -= 1          i -= 1      i = 0      for i in **range**(0, **len**(array)):          array[i] = final[i]    def **radixSort**(array):      const = 1      max\_value = **max**(array)      while max\_value / const > 0:  **count\_sort**(array, const)          const = const \* 10  The best case has time complexity : O (n)  The average case has time complexity: O (nk)  The worst case has time complexity : O (nk) |
| Strengths | * It is stable sort * Faster than many algorithms |
| Weaknesses | * Takes more memory * Difficult to code * Depends on counting sort |
| **Algorithm** | **Insertion Sort** |
| Description | In this type of sorting, the first element in array is taken as sorted and second element is taken as ‘key’ and it is compared with every element in array to place them in their correct position. It is repeated until array is sorted. |
| Pseudo code | Insertion\_Sort(A):  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 |
| Code | def insertSort(A):  n = len(A)  for i in range(1,n):  key = A[i]  j = i-1  while j>=0 and A[j] > key:  A[j+1] = A[j]  j = j-1  A[j+1] = key  #insertion sort in descending order  def insertSort(A):  n = len(A)  for i in range(1,n):  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 | Insertion\_Sort(A):  for j = 2 to A.length n  key = A[j] n-1  i = j – 1 n-1  while i > 0 and A[i] > key  A[i+1] = A[i]  i = i-1 –1)  A[i+1] = key n - 1  The best case has time complexity : O (n)  The average case has time complexity: O (n^2)  The worst case has time complexity : O (n^2) |
| Proof of correctness | Loop invariant  **Step 1 : Initialization**  We begin by showing that the loop invariant holds before the first loop iteration, when j = 2. Subarray A [1 .. j-1], therefore, contains only one element A [1], which is actually the first element of A [1]. In addition, this subarray is sorted, indicating that the invariant loop holds before the first loop iteration.  **Step 2: Maintenance:**  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[j..1] then consists of the elements originally in A[j..1] , but in sorted order.  **Step 3 : Termination:**  The condition causing the for loop to terminate is that j > A:length = n. 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. |
| Strengths | * It is a stable sort algorithm * It works fast on small input array (O(n)) |
| Weaknesses | * It does not perform well on large input array * It takes (O(n^2)) time to perform sorting on large input array |
|  |  |
| **Algorithm** | **Selection sort** |
| Description | In this type of sorting, we repeatedly search for smallest element in array and locate it in the beginning of array. |
| Pseudo code | Selection\_sort(A):  for i=0 to A.length:  minvalue\_index = i  for j in range(i + 1,n):  if A[j] < A[minvalue\_index]  minvalue\_index = j  swap (min\_element, first unsorted position)  end Selection\_sort |
| Code | def selectionSort(A):  n = len(A)  for i in range(n-1):  minValueIndex = i  for j in range(i + 1,n):  if A[j] < A[minValueIndex] :  minValueIndex = j  if minValueIndex != i :  temp = A[i]  A[i] = A[minValueIndex]  A[minValueIndex] = temp  return A  # selection sort in descending order  def selectionSort(A):  n = len(A)  for i in range(n-1):  minValueIndex = i  for j in range(i + 1,n):  if A[j] > A[minValueIndex] :  minValueIndex = j  if minValueIndex != i :  temp = A[i]  A[i] = A[minValueIndex]  A[minValueIndex] = temp  return A |
| Time complexity | def selectionSort(A):  n = len(A) 1  for i = 1 to n-1 : n  minValueIndex = i n - 1  for j = i + 1 to n:  if A[j]>A[minValueIndex]:  minValueIndex = j  if minValueIndex !=i: n - 1  temp = A[i] n - 1  A[i] =A[minValueIndex] n - 1  A[minValueIndex] = temp n - 1  The best case has time complexity : O (n^2)  The average case has time complexity: O (n^2)  The worst case has time complexity : O (n^2) |
| Proof of correctness | Loop invariant  **Step 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 line 3 of the code sets minValueindex = 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.  **Step 2: Maintenance**  Before pass j, we assume that min indexes the smallest element in the subarray A[i..j-1]. During iteration A[j] ≥ A[min],the if statement is not true, so nothing is executed. But now min indexes the smallest element of A[i..j]. Line 6 switches min to index this new location and hence after the loop iteration finishes, min indexes the smallest element in subarray A[i..j].  **Step 3: 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 | * It performs well on small size arrays. * This algorithm does not require a lot of space for sorting. Only one extra variable is used to store temp value. |
| Weaknesses | * It does not perform well on large size array. * It takes O(n^2) time to perform sorting. |
|  |  |
| **Algorithm** | **Quick sort** |
| Description | Quick sort also uses divide and conquer rule. It is similar to merge sort. In this algorithm, we first select pivot and on the basis of pivot, we divide smaller and larger elements to their correct side. |
| Pseudo code | def quickSort(arr[], low, high)  if (low < high)  pi = partition(arr, low, high);  quickSort(arr, low, pi - 1)  quickSort(arr, pi + 1, high)  def partition (arr[], low, high)  pivot = arr[high];  i = (low - 1)  for j = low to (high – 1)  if (arr[j] < pivot)  i++  swap arr[i] and arr[j]  swap arr[i + 1] and arr[high])  return (i + 1) |
| Code | def quick\_sort(A,p,r):  if p < r:  pi = partition(A,p,r)  quick\_sort(A,p,pi-1)  quick\_sort(A,pi+1,r)    def partition(A,p,r):  x = A[right]  i = p - 1  for j = p to r:  if (A[j] < x):  i +=1  temp = A[i]  A[i] = A[j]  A[j] = temp  temp\_2 = A[i+1]  A[i+1] = A[r]  A[r] = temp\_2  return (i+1) |
| Time complexity | def quick\_sort(A,p,r):  if p < r:  pi = partition(A,p,r)  quick\_sort(A,p,pi-1)  quick\_sort(A,pi+1,r)    def partition(A,p,r):  x = A[right]  i = p - 1  for j = p to r:  if (A[j] < x):  i +=1  temp = A[i]  A[i] = A[j]  A[j] = temp  temp\_2 = A[i+1]  A[i+1] = A[r]  A[r] = temp\_2  return (i+1)  The best case has time complexity : O (n.log n)  The average case has time complexity: O (n.log n)  The worst case has time complexity : O (n^2) |
| Proof of correctness | Loop invariant  **Step 1: 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.  **Step 2: Maintenance:**  We consider two cases, depending on the outcome of the test. When A[j] > x; the only action in the loop is to increment j . After j is incremented, condition 2 holds for A[j – 1] and all other entries remain unchanged. When A[j] < x; the loop increments i, swaps A[i] and A[j] , and then increments j . 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.  **Step 3 : 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 | * It is also in-place algorithm. * It works very well on short sized arrays |
| Weaknesses | * It is not a stable sort algorithm * It takes much time on large size arrays which is O(n^2) |
| **Algorithm** | **Counting sort** |
| Description | This sorting works by iterating through array, counting the repitition of every element present in array and use these counts to compute an element index in the final sorted array. |
| Pseudo code | Counting\_sort(A,B,k)  Let C[0 .. k] be a new array  for i = 0 to k  C[i] = 0  for j = 1 to A.length  C[A[j]] +=1  for i = 1 to k  C[i] += C[i-1]  for j = A.length **downto** 1  B[C[A[j]]] = A[j]  C[A[j]] -= 1 |
| Code | def countSort(A):  ans = []  smallest = min(A)  smallest = abs(smallest)  for i in range(0,len(A)):  A[i] += smallest  largest = max(A)  output = [0] \* len(A)  count = [0] \* (largest+1)  for i in range(0,len(A)):  j = A[i]  count[j] +=1  for i in range(1,largest+1):  count[i] += count[i-1]  for i in reversed(range(0,len(A))):  j = A[i]  count[j] -=1  output[count[j]] = A[i]  output[count[j]] -= smallest  print(output) |
| Time complexity | def countSort(A):  ans = [] 1  smallest = min(A) 1  smallest = abs(smallest) 1  for i in range(0,len(A)): n+1  A[i] += smallest n  largest = max(A) 1  output = [0] \* len(A) 1  count = [0] \* (largest+1) 1  for i in range(0,len(A)): n+1  j = A[i] n  count[j] +=1 n  for i in range(1,largest+1): n+1  count[i] += count[i-1] n  for i in reversed(range(0,len(A))): n+1  j = A[i] n  count[j] -=1 n  output[count[j]] = A[i] n  output[count[j]] -= smallest n  print(output)1  The best case has time complexity: O (n + k)  The average case has time complexity: O (n + k)  The worst case has time complexity: O (n + k) |
| Proof of correctness | Loop invariant  **Step 1: 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.  **Step 2: Maintenance:**  We consider two cases, depending on the outcome of the test. When A[j] > x; the only action in the loop is to increment j . After j is incremented, condition 2 holds for A[j – 1] and all other entries remain unchanged. When A[j] < x; the loop increments i, swaps A[i] and A[j] , and then increments j . 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.  **Step 3: 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 | * It is a stable sort algorithm * Since it is not a comparison-based algorithm |
| Weaknesses | * If range of input value is much larger then this algorithm requires a very large space. |
| **Algorithm** | **Shell sort** |
| Description | It is type of sorting which first sorts elements that are far apart from each other and the interval between elements to be sorted is continuously reduced. The interval between the elements is reduced based on the sequence used. |
| Pseudo code | def shellSort()  A : array of items  interval = A.length // 2  while interval > 0 do:  for i = interval to n:  valueToInsert = A[i]  inner = i;  while inner >= interval && A[inner - interval] > valueToInsert do:  A[inner] = A[inner - interval]  inner = inner - interval  end while  A[inner] = valueToInsert  interval = interval //2; |
| Code | def shellSort(array, n):  interval = n // 2  while interval > 0:  for i in range(interval, n):  temp = array[i]  j = i  while j >= interval and array[j - interval] > temp:  array[j] = array[j - interval]  j -= interval  array[j] = temp  interval //= 2 |
| Time complexity | def shellSort(array, n):  interval = n // 2  while interval > 0:  for i in range(interval, n):  temp = array[i]  j = i  while j >= interval and array[j - interval] > temp:  array[j] = array[j - interval]  j -= interval  array[j] = temp  interval //= 2  The best case has time complexity: O (n.log n)  The average case has time complexity: O (n.log n)  The worst case has time complexity: O (n^2) |
| Proof of correctness | Loop invariant  **Step 1: 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.  **Step 2: Maintenance:**  We consider two cases, depending on the outcome of the test. When A[j] > x; the only action in the loop is to increment j . After j is incremented, condition 2 holds for A[j – 1] and all other entries remain unchanged. When A[j] < x; the loop increments i, swaps A[i] and A[j] , and then increments j . 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.  **Step 3: 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 | * It is a stable sort algorithm * Since it is not a comparison-based algorithm |
| Weaknesses | * If range of input value is much larger then this algorithm requires a very large space. |
| **Algorithm** | **Heap Sort** |
| Description | It is also a comparison-based sorting algorithm. It is improved version of selection sort in which we first divide array and find largest element from unsorted array and put it in sorted array. |
| Pseudo Code | heap\_Sort(A):  A.heap-size = A:length  for i = [A.length/2] downto 1  MAX-HEAPIFY(A,i)  MAX-HEAPIFY(A,i)  l = left(i)  r = right(i)  if l <= A.heap-size and A[l] > A[i]  largest = l  else  largest = i  if r <= A.heap-size and A[r] > A[largest]  largest = r  if largest != i  exchange A[i] with A[largest]  MAX-HEAPIFY(A,largest) |
| Code | def heapify(arr, n, i):  largest = i  left = 2 \* i + 1  right = 2 \* i + 2  if left < n and arr[i] < arr[l]:  largest = left  if right < n and arr[largest] < arr[r]:  largest = right  if largest != i:  arr[i],arr[largest] = arr[largest],arr[i]  heapify(arr, n, largest)  def heapSort(arr):  n = len(arr)  for i in range(n // 2 - 1, -1, -1):  heapify(arr, n, i)  for i in range(n-1, 0, -1):  arr[i], arr[0] = arr[0], arr[i]  heapify(arr, i, 0) |
| Time Complexity | def heapify(arr, n, i):  largest = i 1  left = 2 \* i + 1 1  right = 2 \* i + 2 1  if left < n and arr[i] < arr[l]: 1  largest = left 1  if right < n and arr[largest] < arr[r]: 1  largest = right 1  if largest != i: 1  arr[i],arr[largest] = arr[largest],arr[i] 1  heapify(arr, n, largest) 1  def heapSort(arr):  n = len(arr) 1  for i in range(n // 2 - 1, -1, -1): (n/2)+1  heapify(arr, n, i) (n/2)  for i in range(n-1, 0, -1): (n)  arr[i], arr[0] = arr[0], arr[i] n-1  heapify(arr, i, 0) n-1 |
| Proof of correctness | **Initialization:** Prior to the first iteration of the loop, i = [n/2]. Each node [n/2] + 1; [n/2]+2 … n is a leaf and is thus the root of a trivial max-heap.  **Maintenance:** To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A,i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes i + 1; i + 2,…, n are all roots of max-heaps. Decrementing i in the for loop update reestablishes the loop invariant for the next iteration.  **Termination:** At termination, i = 0. By the loop invariant, each node 1,2, …. , n is the root of a max-heap. In particular, node 1 is. |
| Strengths | It takes n.lg(n) time for any input size array. |
| Weakness | It is not a stable sort algorithm |

Table 5: Sorting algorithms with detailed analysis

## 7.1. Integration

First of all, we designed the main window, sorting window and filter window as **.ui** file by using **PyQt5 designer**. Then we converted these **.ui** files to **.py** files by using **pyuic5** command to link these files and add functionalities to them. In the .py file of main window, we use **QThread** as it allows us to scrap the data as well as use other functionalities defined in the file at the same time. After completion of scrapping, data was stored in csv file so we load that from that csv file and displayed it on table widget of designer. For every column, we designed the sorting and filter button. To open sorting or filter window, we press their respective button and link these window files to main window .py file. For this we create object of these sub windows in main window .py file. In sorting window there were two combo boxes, one showing sorting algorithm and other showing ascending/descending option. By pressing button, we get sorting type and algorithm in main .py file and perform sorting on the respective column according to the provided algorithm. Same for the searching, there was one combo box, label and a button. By pressing the button, searching algorithm searched the required entity and then displayed it on the table widget. For the filter window, there were two textboxes in which first textbox get first letter and second textbox get last letter of the entity and search it from the respective column and displayed on the table, if found.

## 7.2. Final user interface

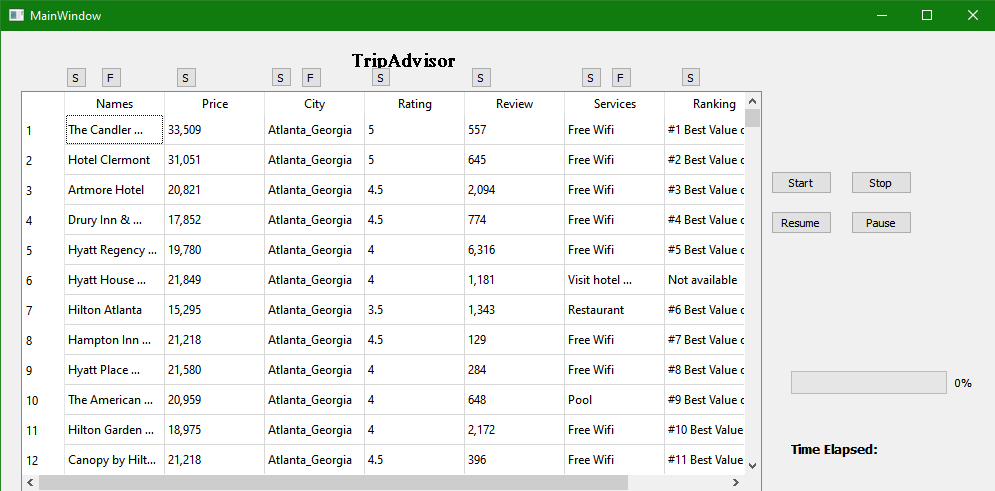


Figure 2: User Interface of main window

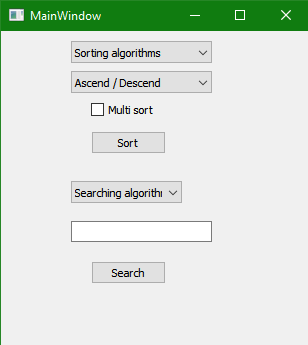


Figure 3: User Interface of searching and sorting window

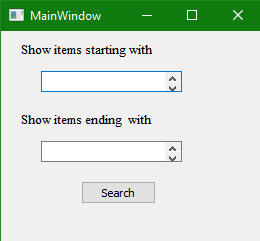


Figure 4:User Interface of Filter window

## 7.3. Collaboration

Both the collaborators have put up the best possible performance in every possible aspect. This project gave us many hurdles during its roadmap but we tried our very best to tackle every problem. Some of the problems we faced are as follows:

* Some of the sorting algorithms were not working as intended. The debugging of the algorithms took time and we completed majority of them
* The integration of codes with the UI was quite difficult. The table widget was not putting up the sorted data. So, we had to make an object of other window in the main window which solved the problem
* Another problem we faced during the integration was that the other attributes of the data were not getting sorted according to the selected data type. To solve this problem, we re-introduced OOP concepts. We passed the data from the classes and the data was sorted as required.
* The searching algorithms were not very hard to implement but the search filter was a bit hard to implement as the algorithm was showing repetitive data.
* Another problem we faced was that the Main window of UI and the scrapping process were not working side by side. So, we had to use multi-threading in which both the processes started to work at same time.