Data Structures and Algorithms

CS-261- Mid Project Final Report

Project Supervisor

Mr. Samyan Qayyum Wahla

Project Members (Project ID 08)

Muneeb Ur Rehman	2019-CS-133
Muhammad Ali Murtaza	2020-CS-114

Department of Computer Science

Table of Contents

Table of Figures

1. Overview

Getting the large amount of data at a single place is the great requirement in this era. Rather it would be collected from different places or different platforms, it doesn't matter. This project solves this problem of managing the large amount of data for the user. GitHub user's specific attributes were collected and saved in this project for the end user.

1.1 Description

GitHub is the most popular platform among the developers around the whole world. Every developer makes his/her account on GitHub and work on it. So, for developers GitHub accounts are very important. For code versions looking, global projects, industrial work whole work on GitHub. So, GitHub power and popularity can not be denied. If anyone wants to know about any person's skills just need to see his/her GitHub profile. So, any organization can hire new engineers just by looking their GitHub profile. So, getting GitHub profile on web site is a very very lengthy process that can not be turned physical. So, to turn this physical or be the practical solves many problems of the organizations.

Scrapping is the process of getting data of from web site and save it on local storage. Above mentioned problem can be solved through one of the problems by scrapping. So, scrapping of GitHub user profile is the aim of this project to solve the above problems and present required information at ease and on time.

1.2 Motivation

Problems are the only sources in the world that gave us the solutions. Accessing the large amount of GitHub user data to analyze for sorting or comparing the users is the motivation for this project. This project is the important solution for organizations that need to hire developers.

1.3 Audience

The main audience are

- Computer Science students
- Global Industry
- Research Institutes

Anyone who is related to hiring software developers.

1.4 Business Need

Business need defines the financial needs by this project for competition. This project needs just the good internet connection and the system on which the code will execute. All the scrapped information is saved on the system local disk. Data requirement amount defines the disk space. By the way not enough space required. It can be done in few MBs of space.

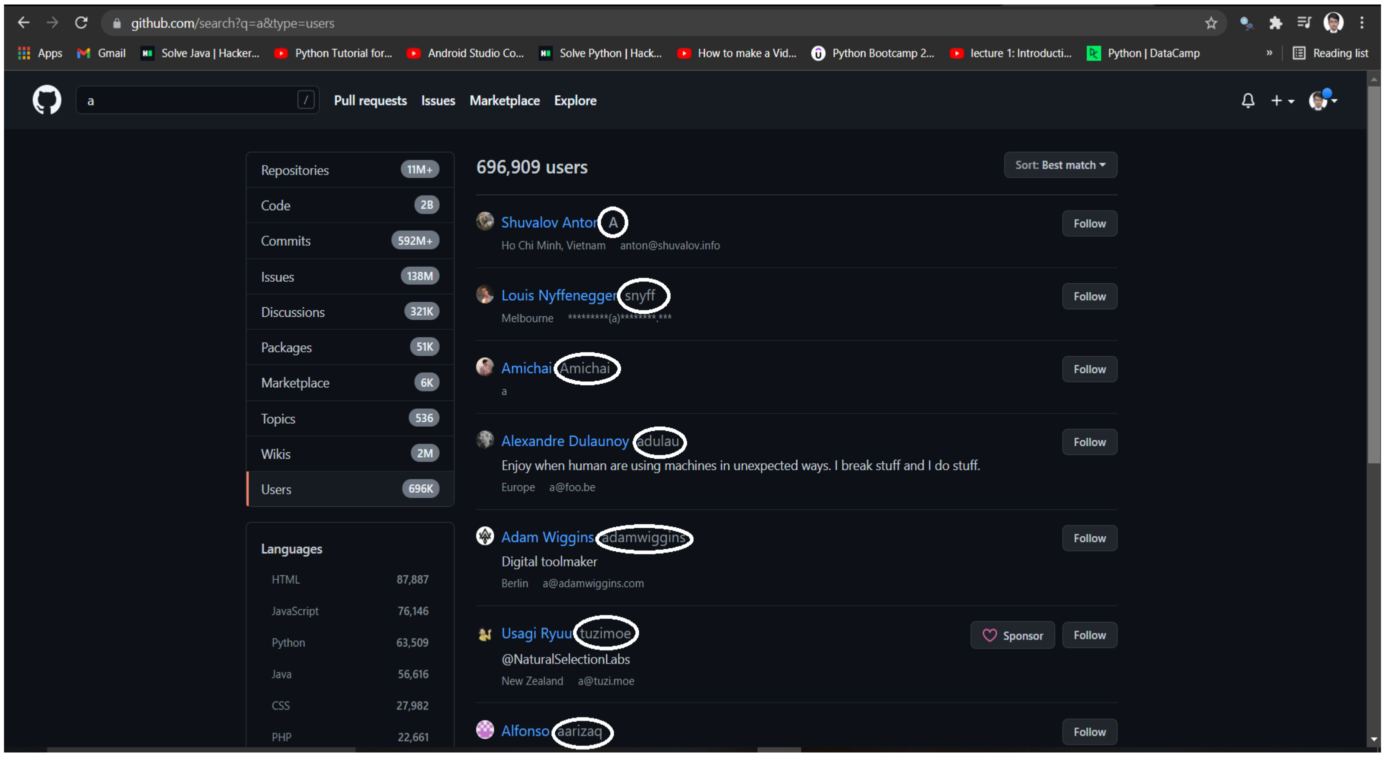
2. Scraping

Web Scrapping is the extraction of data from the web sites and save it in the required format on the local storage. Scarping is the process done due to observe and work on large amount of data present on web sites.

2.1 Source

In scrapping sources could be the sources. It could be possible for the situation that the user scraps the same attributes from the different websites. But in out case, our source is same. Our base link is same among the whole project that is github.com

Two times scrapping is required for this project. One is got to scrap just user name. Its scrapping window is…


github.com/search?q=a&type=users

Apps Gmail Solve Java | Hacker... Python Tutorial for... Android Studio Co... Solve Python | Hack... How to make a Vid... Python Bootcamp 2... lecture 1: Introducti... Python | DataCamp Reading list

a Pull requests Issues Marketplace Explore

Repositories	11M+
Code	2B
Commits	592M+
Issues	138M
Discussions	321K
Packages	51K
Marketplace	6K
Topics	536
Wikis	2M
Users	696K

Languages

HTML	87,887
JavaScript	76,146
Python	63,509
Java	56,616
CSS	27,982
PHP	22,661

696,909 users Sort: Best match

Shuvalov Anton A — Ho Chi Minh, Vietnam anton@shuvalov.info — Follow

Louis Nyffenegger snyff — Melbourne *********(a)********.*** — Follow

Amichai Amichai — a — Follow

Alexandre Dulaunoy adulau — Enjoy when human are using machines in unexpected ways. I break stuff and I do stuff. — Europe a@foo.be — Follow

Adam Wiggins adamwiggins — Digital toolmaker — Berlin a@adamwiggins.com — Follow

Usagi Ryuu tuzimoe — @NaturalSelectionLabs — New Zealand a@tuzi.moe — Sponsor Follow

Alfonso aarizaq — Follow

Figure 1: User Name Scrapping Window

These are all highlighted user names in the above window. These are scrapped first and saved in the csv simple file. This process could be making the automatic or at code level by creating a query at code level. It means that we need about 1 million data, but this page contains only 10 names. What about the others? GitHub contains about 56 million data. Where is that huge amount of data? How could I access that data? This all the data could be accessed through queries. Now queries are the new link that your code generate to go to the next page. Pagination is included in this query generating. A simple sample is given in the figure below:

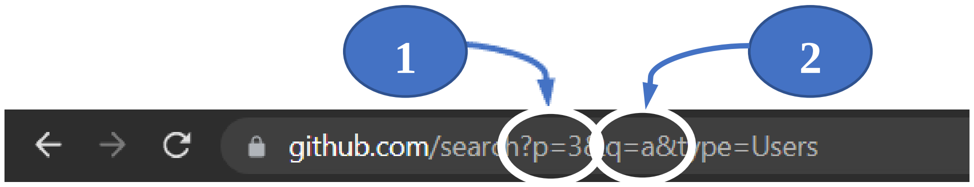


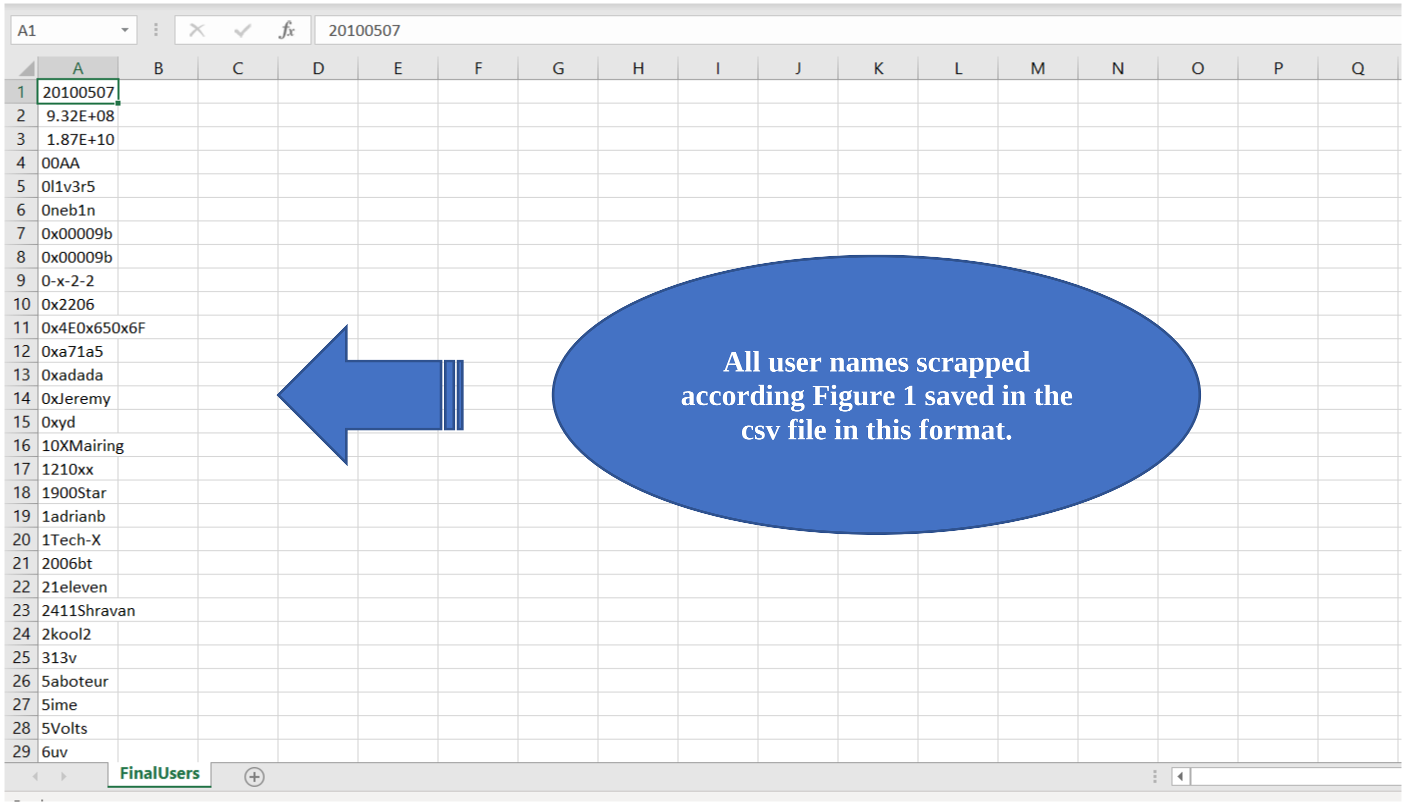
Figure 2: Query and Pagination

The above figure, figure 2 numbers are explained in the following table:

Sr.	Description
1.	Number 1 label in the figure 2 describe pages. Here the example of ‘p=3’, means that this query is of page number 3 of the web site. Here actually described the concept of pagination, where query is same as ‘q=a’ and for this same query different pages link 1, 2, 3 are accessed.
2.	Label 2 is pointing to the query which is as ‘q=a’. I mean that the user containing the letter a in their name are searched. Query can be updated or change at code level by giving different letters at code like a, b, c ….

	This will change the link and you can search different users in the same code running by run time change of query.

After completion of the process of query handling and pagination, a simple csv file generated in which all scrapped user name is saved in a simple format that is shown below:



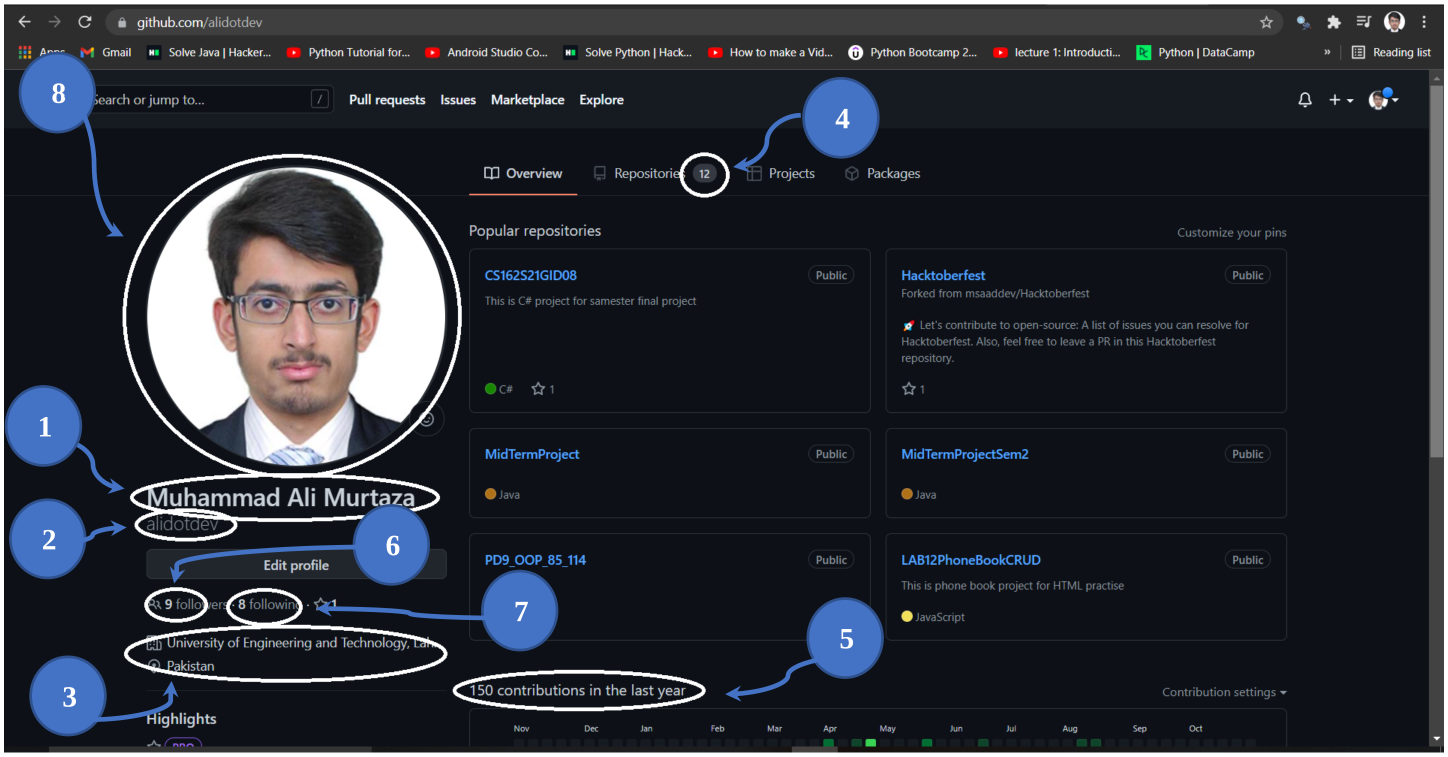
A1 | 20100507

	A
1	20100507
2	9.32E+08
3	1.87E+10
4	00AA
5	0l1v3r5
6	0neb1n
7	0x00009b
8	0x00009b
9	0-x-2-2
10	0x2206
11	0x4E0x650x6F
12	0xa71a5
13	0xadada
14	0xJeremy
15	0xyd
16	10XMairing
17	1210xx
18	1900Star
19	1adrianb
20	1Tech-X
21	2006bt
22	21eleven
23	2411Shravan
24	2kool2
25	313v
26	5aboteur
27	5ime
28	5Volts
29	6uv

All user names scrapped according Figure 1 saved in the csv file in this format.

Figure 3: Scrapped Username csv File

For actual scrapping another window is used that obviously increases the cost and time of the project. But it necessary because GitHub is a secure website that does not allow anyone to scrap its user's data. But it's done by applying this extra cost. Scrapping window of profile is given below…



CS162S21GID08 Public
This is C# project for samester final project
C# 1

Hacktoberfest Public
Forked from msaaddev/Hacktoberfest
Let's contribute to open-source: A list of issues you can resolve for Hacktoberfest. Also, feel free to leave a PR in this Hacktoberfest repository.
1

MidTermProject Public
Java

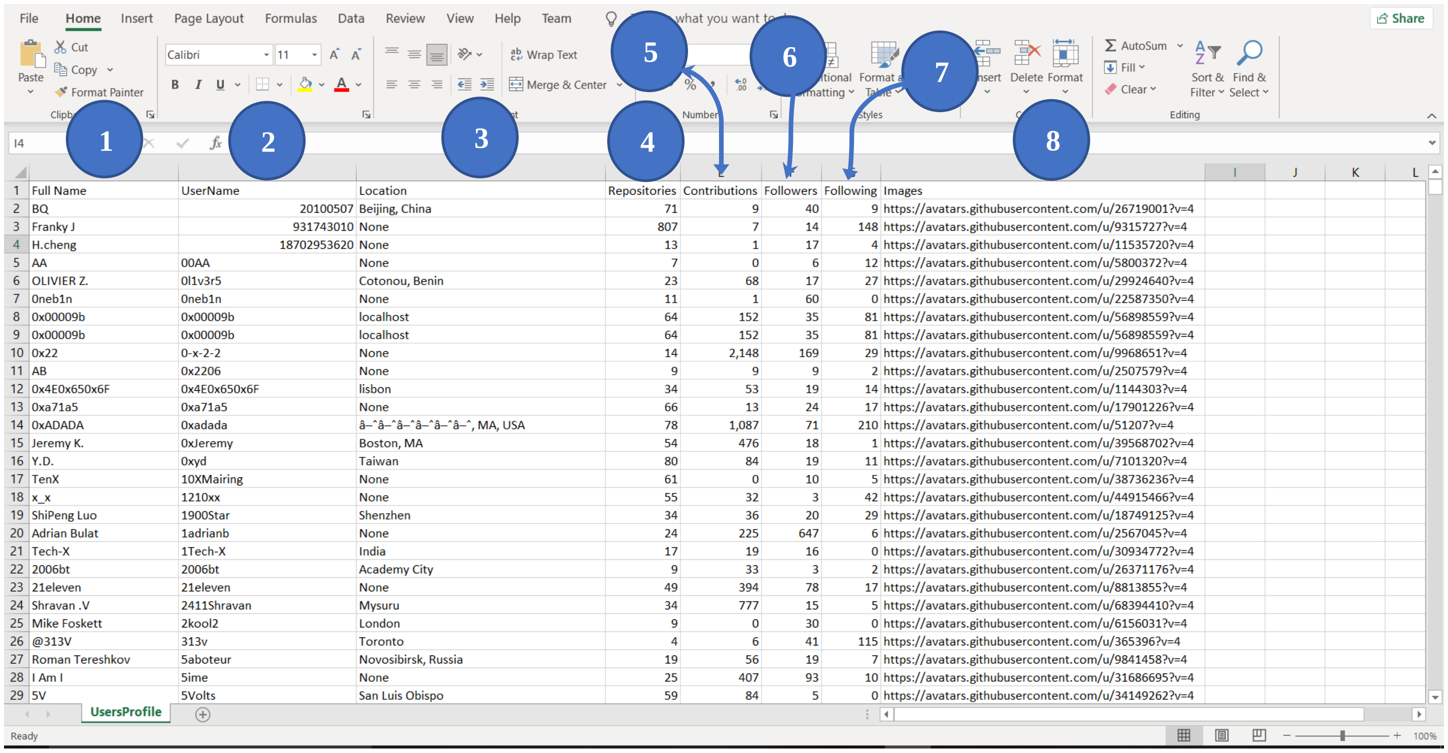
MidTermProjectSem2 Public
Java

PD9_OOP_85_114 Public

LAB12PhoneBookCRUD Public
This is phone book project for HTML practise
JavaScript

Figure 4: Profile Scrapping Window

Figure 4 described the attributes location from the web site. After scrapping from web site, generated csv file is shown in which attributes are shown with the same numbering:



	Full Name	UserName	Location	Repositories	Contributions	Followers	Following	Images
2	BQ	20100507	Beijing, China	71	9	40	9	https://avatars.githubusercontent.com/u/26719001?v=4
3	Franky J	931743010	None	807	7	14	148	https://avatars.githubusercontent.com/u/9315727?v=4
4	H.cheng	18702953620	None	13	1	17	4	https://avatars.githubusercontent.com/u/11535720?v=4
5	AA	00AA	None	7	0	6	12	https://avatars.githubusercontent.com/u/5800372?v=4
6	OLIVIER Z.	0l1v3r5	Cotonou, Benin	23	68	17	27	https://avatars.githubusercontent.com/u/29924640?v=4
7	0neb1n	0neb1n	None	11	1	60	0	https://avatars.githubusercontent.com/u/22587350?v=4
8	0x00009b	0x00009b	localhost	64	152	35	81	https://avatars.githubusercontent.com/u/56898559?v=4
9	0x00009b	0x00009b	localhost	64	152	35	81	https://avatars.githubusercontent.com/u/56898559?v=4
10	0x22	0-x-2-2	None	14	2,148	169	29	https://avatars.githubusercontent.com/u/9968651?v=4
11	AB	0x2206	None	9	9	9	2	https://avatars.githubusercontent.com/u/2507579?v=4
12	0x4E0x650x6F	0x4E0x650x6F	lisbon	34	53	19	14	https://avatars.githubusercontent.com/u/1144303?v=4
13	0xa71a5	0xa71a5	None	66	13	24	17	https://avatars.githubusercontent.com/u/17901226?v=4
14	0xADADA	0xadada	â–ˆâ–ˆâ–ˆâ–ˆâ–ˆ, MA, USA	78	1,087	71	210	https://avatars.githubusercontent.com/u/51207?v=4
15	Jeremy K.	0xJeremy	Boston, MA	54	476	18	1	https://avatars.githubusercontent.com/u/39568702?v=4
16	Y.D.	0xyd	Taiwan	80	84	19	11	https://avatars.githubusercontent.com/u/7101320?v=4
17	TenX	10XMairing	None	61	0	10	5	https://avatars.githubusercontent.com/u/38736236?v=4
18	x_x	1210xx	None	55	32	3	42	https://avatars.githubusercontent.com/u/44915466?v=4
19	ShiPeng Luo	1900Star	Shenzhen	34	36	20	29	https://avatars.githubusercontent.com/u/18749125?v=4
20	Adrian Bulat	1adrianb	None	24	225	647	6	https://avatars.githubusercontent.com/u/2567045?v=4
21	Tech-X	1Tech-X	India	17	19	16	0	https://avatars.githubusercontent.com/u/30934772?v=4
22	2006bt	2006bt	Academy City	9	33	3	2	https://avatars.githubusercontent.com/u/26371176?v=4
23	21eleven	21eleven	None	49	394	78	17	https://avatars.githubusercontent.com/u/8813855?v=4
24	Shravan .V	2411Shravan	Mysuru	34	777	15	5	https://avatars.githubusercontent.com/u/68394410?v=4
25	Mike Foskett	2kool2	London	9	0	30	0	https://avatars.githubusercontent.com/u/6156031?v=4
26	@313V	313v	Toronto	4	6	41	115	https://avatars.githubusercontent.com/u/365396?v=4
27	Roman Tereshkov	5aboteur	Novosibirsk, Russia	19	56	19	7	https://avatars.githubusercontent.com/u/9841458?v=4
28	I Am I	5ime	None	25	407	93	10	https://avatars.githubusercontent.com/u/31686695?v=4
29	5V	5Volts	San Luis Obispo	59	84	5	0	https://avatars.githubusercontent.com/u/34149262?v=4

Figure 5: Users Profile Excel File

Labeled in the figure 4 and 5 represent to the same type data that is scrapped. Figure 4 just presented scrapping version and figure 5 shown scrapped version of that same data. A table given below show just the name of entities that are numbered in figure 4 and 5.

Numbering	Entities
1	Full Name
2	User Name
3	Location
4	Repositories
5	Contributions
6	Followers
7	Following
8	Image Link

Above mentioned entities are scrapped and saved format in csv file is shown above in figure 5.

2.2 Entities

All the entities are that are scrapped from the web site are shown in the table with their code name and data type…

Number	Entity	Code Name	Data Type
1	Full name of user	Name	String
2	User name of user	UserName	String
3	Location	Location	String
4	Repositories	Repositories	Integer
5	Contributions	Contributions	Integer
6	Followers	Followers	Integer
7	Following	Following	Integer
8	Image	Image	String

3. Features

Features that are offered by this project:

Sr.	Feature	Description
1.	Double Scrapping	Double scrapping process is done in this project. Once the user's name is just scrapped by generating different queries on GitHub web site are stored in csv file.
2.	Profile Scrapping	Profile scrapping of GitHub users through saved user name saved in csv file. Profile scrapping are then scrapped and saved in formatted csv file shown in Figure 4.
3.	Stop Scarping Process	Stops the process of scrapping for the button click.
4.	Pause Scrapping Process	Pause the process of scrapping on clicking the button. Continue button can be used to continue the process of scrapping from where it left.
5.	Show Scrapping Process Progress	Progress bar GUI component is used to show that the scrapping process is in progress or how much profiles are scrapped from the web site.
6.	Multiple Sorting Algos	An option given to the user to sort data through different sorting algorithms.
7.	Saving to csv file	Data scrapped from the scrapping source is saved in the csv file in the required format. Required attributes are saved under required attributes.

3.1 Sorting Data

Sorting is one of the features applied on scrapped data. Sorting algorithms that are used as follows:

- Insertion Sort
- Selection Sort
- Merge Sort
- Bubble Sort
- Count Sort
- Radix Sort
- Bucket Sort
- Hybrid Sort
- Quick Sort
- Shell Sort
- Heap Sort

3.2 Insertion Sort

1. We start by making the second element of the given array
2. We compare the key element with the element(s) before it, in this case, element at index 0:

- If the key element is less than the first element, we insert the key element before the first element.
- If the key element is greater than the first element, then we insert it after the first element.

3. Then, we make the third element of the array as key and will compare it with elements to its left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

3.2.1 Pseudo code:

```
for j <- 2 to length[A]
      do key <- A[j]
        Insert A[j] into the sorted sequence A [1 ⋯ j - 1].
       i <- j - 1
       while i > 0 and A[i] > key
          do A [i + 1] <- A[i]
             i <- i - 1
       A [i + 1] <- key
```

3.2.2 Python code:

```
def insertion_sort(arr):

    for i in range (1, len(arr)):

        value = arr[i]

        j = i - 1
        while j >= 0 and value < arr[j]:
            arr [j + 1] = arr[j]
            j -= 1
        arr [j + 1] = value
    return arr
```

3.2.3 Time Complexity Analysis:

Worst Case Time Complexity **$O(n^2)$**
Best Case Time Complexity **O(n)**
Average Time Complexity **$O(n^2)$**
Space Complexity: **O (1)**

3.2.4 Proof of Correctness:

- **Initialization -** The sub array starts with the first element of the array, and it is (obviously) sorted to begin with.
- **Maintenance -** Each iteration of the loop expands the sub array, but keeps the sorted property. An element gets inserted into the array only when it is greater than the element to its left. Since the elements to its left have already been sorted, it means is greater than all the elements to its left, so the array remains sorted.
- **Termination -** The code will terminate after has reached the last element in the array, which means the sorted sub array has expanded to encompass the entire array. The array is now fully sorted.

3.2.5 Strength:

- The relative order of items with equal keys does not change.
- The ability to sort a list as it is being received.
- Efficient for small data sets, especially in practice than other quadratic algorithms i.e., $O(n^2)$.

3.2.6 Weaknesses:

- Insertion sort is that it does not perform as well as other, better sorting algorithms.
- With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list. Therefore, the insertion sort is particularly useful only when sorting a list of few items.
- One of the major disadvantages of Insertion sort is its Average Time Complexity of O(n^2)

3.2.7 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`

Output `arr = [2, 4, 5, 7, 8, 19, 23]`

3.3 Selection Sort

- Set MIN to location 0
- Search the minimum element in the list
- Swap with value at location MIN
- Increment MIN to point to next element
- Repeat until list is sorted

3.3.1 Pseudo code:

```
for i in 0 to L. length - 1
     min_idx = i
    for j in i+1 to L. length - 1
```

```
        if L[j] < L[min_idx]
           min_idx = j
        endif
    end for

     swap L[i] with L[min_idx]
end for
```

3.3.2 Python code:

```
def selectionSort (array, size):
    for step in range(size):
        min_idx = step

        for i in range (step + 1, size):

            if array[i] < array[min_idx]:
                min_idx = i
                (array [step], array[min_idx]) = (array[min_idx], array[step])
```

3.3.3 Time Complexity Analysis:

Worst Case Time Complexity **O(n^2)**
Best Case Time Complexity **O(n^2)**
Average Time Complexity **O(n^2)**
Space Complexity: **O (1)**

3.3.4 Proof of Correctness:

- **Initialization** - the loop invariant is true at the beginning of the loop. The beginning of the loop is when j = i+1 and min idx = i, so the loop invariant states that L[i] is the smallest element from among L [I … (i+1)] = L[i] which is true.
- **Maintenance** - the body of the loop (including the increment of the loop index j) preserves the invariant. We know that between lines 4 and 5, L[min_idx] is the smallest from among L [I ... j]. The body of the loop checks if L[j] is smaller than L[min_idx] and if so, sets min_idx to j, so between lines 7 and 8 we know that L[min_idx] is the smallest from among L [I ... (j+1)]. When j is incremented when the loop rolls around, the loop invariant will still be true between lines 4 and 5.
- **Termination** - We therefore get to conclude that the invariant is true after the loop is over, which happens when j=len(L). In this case, substituting j=len(L) into the loop invariant, we know that on line 9 L[min_idx] is the smallest from among L [I … len(L)].

3.3.5 Strength:

- Poor efficiency when dealing with a huge list of items.
- The selection sort requires n-squared number of steps for sorting n elements.
- Quick Sort is much more efficient than selection sort

3.3.6 Weaknesses:

- Poor efficiency when dealing with a huge list of items.
- The selection sort requires n-squared number of steps for sorting n elements.
- Quick Sort is much more efficient than selection sort

3.3.7 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`

Output `arr = [2, 4, 5, 7, 8, 19, 23]`

3.4 Merge Sort

1. We start by making the second element of the given array
2. We compare the key element with the element(s) before it, in this case, element at index 0:
If the key element is less than the first element, we insert the key element before the first element.
If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as key and will compare it with elements to its left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

3.4.1 Pseudo code:

```
n1 <-- q - p + 1
n2 <-- r - q
create arrays L [1 ... n1 + 1] and R [1 ... n2 + 1]
for i <-- 1 to n1
      do L[i] <-- A [p + i - 1]
for j <-- 1 to n2
     do R[j] <-- A [q + j]
L [n1 + 1] <-- infinite
R [n2 + 1] <-- infinite
i <-- 1
j <-- 1
for k <-- p to r
    do if L[i] <= R[j]
          then A[k] <-- L[i]
               i <-- i + 1
          else A[k] <-- R[j]
               j <-- j + 1
```

3.4.2 Python code:

```
def MergeSort (num, p, r):
    if (p == r):
        return
    else:
        q = (p+r)//2
        sort1 = MergeSort (num, p, q)
        sort2 = MergeSort (num, q+1, r)
        data = Merge (num, p, q, r)
        return data

def Merge (num, p, q, r):
```

```
n1 = q - p + 1
n2 = r - q
C = []
R = []
L = []
for i in range(p,q+1):
    L.append(num[i])
L.append (9999)
for i in range(q+1,r+1):
   R. append (num[i])

R. append (9999)

j = i = 0

index = p
for s in range((r-p+1)):

    if(L[i] > R[j]):
        num[index] = R[j]
        j += 1
    else:
        num[index] = L[i]
        i += 1
    index += 1

return num
```

3.4.3 Time Complexity Analysis:

Worst Case Time Complexity **O (n log n)**
Best Case Time Complexity **O (n log n)**
Average Time Complexity **O (n log n)**
Space Complexity: **O (n)**

3.4.4 Proof of Correctness:

- **Initialization** - prior to the first iteration of the loop, we have k = p, so that sub array A [p ... k - 1] is empty. this empty sub array contains the k - p = 0 smallest elements of L and R, and since i = j = 1, both L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.
- **Maintenance** - To see that each iteration maintains the loop invariant, let us first suppose that l[i] <= R[j]. Then L[i] is the smallest element not yet copied back into A. Because A [p ... k - 1] contains the k - p smallest elements, after line 14 copies L[i] into A[k], the sub array A [p ... k] will contain the k - p + 1 smallest elements. Incrementing k (in the for loop update) and i(in line 15) re-establishes the loop invariant for the next iteration. If instead L[i] > R[j], then lines 16-17 perform the appropriate action to maintain the loop invariant.
- **Termination** - At termination, k = r + 1. By the loop invariant, the sub array A [p ... k - 1], which is A[p ... r], contains the k - p = r - p + 1 smallest elements of L[1 ... n1 + 1] and R[1 ... n2 + 1], in sorted order. The arrays L and R together contain n1 + n2 + 2 = r - p + 3 elements. All but the two largest elements have been copied back into A, and these two largest elements are the sentinels.

3.4.5 Strength:

- It has a consistent running time, carries out different bits with similar times in a stage.
- Reading of the input during the run-creation step is sequential
- If heap sort is used for the in-memory part of the merge, its operation can be overlapped with I/O

3.4.6 Weaknesses:

- Insertion sort is that it does not perform as well as other, better sorting algorithms.
- With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list. Therefore, the insertion sort is particularly useful only when sorting a list of few items.
- One of the major disadvantages of Insertion sort is its Average Time Complexity of O(n^2).

3.4.7 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`

Output `arr = [2, 4, 5, 7, 8, 19, 23]`

3.5 Bubble Sort

- Starting with the first element (index = 0), compare the current element with the next element of the array.
- If the current element is greater than the next element of the array, swap them.
- If the current element is less than the next element, move to the next element. Repeat Step 1.

3.5.1 Pseudo code:

```
procedure bubbleSort (list: array of items)
   loop = list. count
   for i = 0 to loop-1 do:
      swapped = false

      for j = 0 to loop-1 do:

         if list[j] > list[j+1] then
            /* swap them */
            swap(list[j], list[j+1])
            swapped = true
         end if

      end for

      if (not swapped) then
         break
      end if

   end for
   end procedure return list
```

3.5.2 Python code:

```
def BubbleSort(num):
    n = len(num)
    for i in range(n):

        for j in range (0, n-i-1):

            if(num[j] > num[j+1]):
                num[j+1], num[j] = num[j], num[j+1]

    return num
```

3.5.3 Time Complexity Analysis:

Worst Case Time Complexity **O(n^2)**
Best Case Time Complexity **O(n)**
Average Time Complexity **O(n^2)**

3.5.4 Proof of Correctness:

- **Initialization** - Initially the sub array contains only the last element A[n]A[n] and this is the smallest element of the sub array.
- **Maintenance** - In every step we compare A[j]A[j] with A[j - 1]A[j−1] and make A[j - 1]A[j−1] the smallest among them. So, after the iteration, the length of the sub array increases by one and the first element is the smallest of the sub array.
- **Termination** - The loop terminates when j = i + 1 j=i+1. At that point also the length of the sub array increases by one and the first element is the smallest of the sub array as we swap

3.5.5 Strength:

- It is popular and easy to implement.
- Elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum
- This makes for a very small and simple computer program.

3.5.6 Weaknesses:

- The main disadvantage of the bubble sort is the fact that it does not deal well with a list containing a huge number of items.
- The bubble sort requires n-squared processing steps for every n number of elements to be sorted.
- The bubble sort is mostly suitable for academic teaching but not for real-life applications.

3.5.7 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`
Output `arr = [2, 4, 5, 7, 8, 19, 23]`

3.6 Quick Sort

- Choose the highest index value has pivot-Step
- Take two variables to point left and right of the list excluding pivot Step
- Left points to the low indexStep

- Right points to the highStep
- while value at left is less than pivot move rightStep
- while value at right is greater than pivot move leftStep
- if both step 5 and step 6 does not match swap left and rightStep
- if left ≥ right, the point where they met is new pivot

3.6.1 Pseudo code:

```
QUICKSORT (A, p, r)
1 if p < r
2 then q ← PARTITION (A, p, r)
3 QUICKSORT (A, p, q-1)
4 QUICKSORT (A, q+1, r)

PARTITION (A, p, r)
1 x ← A[r]
2 i ← p - 1
3 for j ← p to r-1
4 do if A[j] ≤ x
5 then i ← i + 1
6 exchange A[i] ↔ A[j]
7 exchange A[i+1] ↔ A[r]
8 return i+1
```

3.6.2 Python code:

```
def partition (arr, low, high):
    i = (low-1)
    pivot = arr[high]

    for j in range (low, high):

        if arr[j] <= pivot:

            i = i+1
        arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)

def quickSort (arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:

        pi = partition (arr, low, high)
    quickSort (arr, low, pi-1)
    quickSort (arr, pi+1, high)
```

3.6.3 Time Complexity Analysis:

Worst Case Time Complexity **O (n^2)**

Best Case Time Complexity **O (n log n)**
Average Time Complexity **O (n log n)**

3.6.4 Proof of Correctness:

- **Initialization** - Prior to the first iteration of the loop, i = p - 1, and j = p. There are no values between p and i, and no values between i+1 and j-1, so the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition
- **Maintenance** - There are two cases to consider depending on the outcome of the test in line 4: • When A[j] > x, the only action in the loop is to increment j. After j is incremented, condition 2 holds for all A[j-1] and all other entries remain unchanged. • When A[j] ≤ x, i is incremented, A[i] and A[j] are swapped, and then j is incremented. Because of the swap, we now have that A[i] ≤ x, and condition 1 is satisfied. Similarly, we also have that A[j-1]>x, since the item that was swapped into A[j-1] is, by the loop invariant, greater than x.
- **Termination** - At termination, j = r. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

3.6.5 Strength:

- Its average-case time complexity to sort an array of n elements is O (n lg n).
- On the average it runs very fast, even faster than Merge Sort.
- It requires no additional memory

3.6.6 Weaknesses:

- Its running time can differ depending on the contents of the array.
- Quick sort running time degrades if given an array that is almost sorted (or almost reverse sorted). Its worst-case running time, O (n^2) to sort an array of n elements, happens when given a sorted array.
- It is not stable.

3.6.7 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`
Output `arr = [2, 4, 5, 7, 8, 19, 23]`

3.7 Heap Sort

- Construct a Binary Tree with given list of Elements.
- Transform the Binary Tree into Min Heap.
- Delete the root element from Min Heap using Happify method.
- Put the deleted element into the Sorted list.
- Repeat the same until Min Heap becomes empty.
- Display the sorted list.

3.7.1 Pseudo code:

```
procedure heapify (a, count) is
    (end is assigned the index of the first (left) child of the root)
    end := 1

    while end < count
```

```
        siftUp (a, 0, end)
        end:= end + 1
procedure siftUp (a, start, end) is
    child := end
    while child > start
        parent := iParent(child)
        if a[parent] < a[child] then (out of max-heap order)
            swap(a[parent], a[child])
            child := parent (repeat to continue sifting up the parent now)
        else
            return
```

3.7.2 Python code:

```
def heapSort (arr):
    n = len(arr)
    for i in range (n//2 - 1, -1, -1):
        heapSort (arr, n, i)

    for i in range (n-1, 0, -1):
        arr[i], arr [0] = arr [0], arr[i]
        heapSort (arr, i, 0)
```

3.7.3 Time Complexity Analysis:

Worst Case Time Complexity **O (n log n)**
Best Case Time Complexity **O (n log n)**
Average Time Complexity **O (n log n)**

3.7.4 Proof of Correctness:

- **Initialization -** Prior to the first iteration. Everything is a leaf so it is already a heap.
- **Maintenance -** Let us assume that we have a working solution till now. The children of node i are numbered higher than i. MaxHeapify preserves the loop invariant as well. We maintain the invariance at each step.
- **Termination -** Terminates when the i drops down to 0 and by the loop invariant, each node is the root of a max-heap.

3.7.5 Strength:

- The Heap sort algorithm is widely used because of its efficiency.
- The Heap sort algorithm can be implemented as an in-place sorting algorithm
- Its memory usage is minimal.

3.7.6 Weaknesses:

- Heap sort requires more space for sorting
- Quick sort is much more efficient than Heap in many cases
- Heap sort make a tree of sorting elements.

3.7.7 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`
Output `arr = [2, 4, 5, 7, 8, 19, 23]`

3.8 Shell Sort

- Initialize the value of h
- Divide the list into smaller sub-list of equal interval h
- Sort these sub-lists using insertion sort
- Repeat until complete list is sorted

3.8.1 Pseudo code:

```
procedure shellSort ()
   A: array of items

     while interval < A. length /3 do:
      interval = interval * 3 + 1
   end while

   while interval > 0 do:

      for outer = interval; outer < A. length; outer ++ do:

        valueToInsert = A[outer]
      inner = outer;


         while inner > interval -1 && A [inner - interval] >= valueToInsert do:
            A[inner] = A [inner - interval]
            inner = inner - interval
         end while
      A[inner] = valueToInsert

      end for

     interval = (interval -1) /3;

   end while
   end procedure
```

3.8.2 Python 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
    return num
```

3.8.3 Time Complexity Analysis:

Worst Case Time Complexity **O(n^2)**
Best Case Time Complexity **O (n log n)**
Average Time Complexity **O (n log n)**

3.8.4 Proof of Correctness:

Shell sort has an outer loop that determines a sequence of gaps; that sequence ends with 1.
When the gap is equal to 1, the two inner loops become equivalent to an insertion sort. So, the fact that shell sort is a valid sorting algorithm relies on these facts:
In the initial iterations, where the gap is larger than one, the algorithm only permutes the input array; it does not add, remove or modify keys. (This is easy to prove directly from the program code.)
The result of sorting an array is the same as the result of sorting a permutation of the array. (You could prove this mathematically but it's quite obvious.)
In the last iteration, where the gap is equal to one, the algorithm reduces to insertion sort. (This is also easy to prove directly from the program code, by substituting the value 1 for the gap and comparing to the code for insertion sort.)
Insertion sort is a valid sorting algorithm. (This requires a proof but we've now got rid of all the complication with the gaps.)
This is the argument I'd use. Remains to show that insertion sort works properly. To prove that, you can use loop invariants:
In the outer loop of insertion sort, that iterates over an index i, we have the invariant that everything at indices lower than i is in sorted order. We have to make sure that this invariant is preserved over each iteration.
In the inner loop of insertion sort, which has an index j that counts down from i to the position where the key needs to be inserted, we have the invariants that:
All the keys at indices smaller than j are in sorted order
All the keys at indices in between j and i are in sorted order and larger than the key.

3.8.5 Strength:

- Shell sort algorithm is only efficient for finite number of elements in an array.
- Shell sort algorithm is 5.32 x faster than bubble sort algorithm.
- It sorts in place, no additional storage is required as well

3.8.6 Weaknesses:

- Shell sort algorithm is complex in structure and bit more difficult to understand.
- Shell sort algorithm is significantly slower than the merge sort, quick sort and heap sort algorithms.
- Unstable sorting algorithm because this algorithm does not examine the elements lying in between the intervals.

3.8.7 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`
Output `arr = [2, 4, 5, 7, 8, 19, 23]`

3.9 Hybrid Sort

This sorting algorithm is hybrid of **QUICK** sort and **Insertion** sort. So that it combines the positive points of both algorithms.

3.9.1 Pseudo code:

```
HYBRIDSORT (A, p, r, k)
if (p < r)
    if (r - p + 1 > k)
        q = PARTITION (A, p, r)
        HYBRIDSORT (A, p, q, k)
        HYBRIDSORT (A, q + 1, r, k)

INSERTIONSORT(A)
```

3.9.2 Python code:

```
def HYBRIDSORT (A, p, r, k):
    if (p < r):
        if (r - p + 1 > k):
            q = PARTITION (A, p, r)
            HYBRIDSORT (A, p, q, k)
            HYBRIDSORT (A, q + 1, r, k)

    InsertionSort(A)
```

3.9.3 Time Complexity Analysis:

Time Complexity: **O(n^2)**
Auxiliary Space: **O(n)**

3.9.4 Strength:

- Its average-case time complexity to sort an array of n elements is O (n lg n).
- On the average it runs very fast, even faster than Merge Sort.
- It requires no additional memory

3.9.5 Weaknesses:

- Its running time can differ depending on the contents of the array.
- Quick sort running time degrades if given an array that is almost sorted (or almost reverse sorted). Its worst-case running time, O (n^2) to sort an array of n elements, happens when given a sorted array.
- It is not stable.

3.9.6 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`
Output `arr = [2, 4, 5, 7, 8, 19, 23]`

3.10 Count Sort

- Find out the maximum element from the given array.
- Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.
- Store the count of each element at their respective index in count array

- Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.
- Find the index of each element of the original array in the count array. This gives the cumulative count.
- After placing each element at its correct position, decrease its count by one.

3.10.1 Pseudo code:

```
CountingSort(A)
for i = 0 to k do
    c[i] = 0
for j = 0 to n do
    c[A[j]] = c[A[j]] + 1
for i = 1 to k do
    c[i] = c[i] + c[i-1]
for j = n-1 downto 0 do
    B [ c[A[j]]-1] = A[j]
    c[A[j]] = c[A[j]] - 1
end func
```

3.10.2 Python code:

```python
def countingSort(array):
    size = len(array)
    output = [0] * size

    count = [0] * 10

    for i in range (0, size):
        count[array[i]] += 1

    for i in range (1, 10):
        count[i] += count [i - 1]

    i = size - 1
    while i >= 0:
        output[count[array[i]] - 1] = array[i]
        count[array[i]] -= 1
        i -= 1

    for i in range (0, size):
        array[i] = output[i]
```

3.10.3 Time Complexity Analysis:

Worst Case Time Complexity **O (n + k)**
Best Case Time Complexity **O(n)**
Average Time Complexity **O (n + k)**

3.10.4 Strength:

- The biggest advantage of counting sort is its complexity O(n+k), where n is the size of the sorted array and k is the size of the helper array.
- Since it is not a comparison-based sorting, it is not lower bounded by O (n log n) complexity.

- Reduced space complexity if the range of elements is narrow, that is, more frequency of close integers.

3.10.5 Weaknesses:

- Counting sort only works when the range of potential items in the input is known ahead of time.
- Space cost. If the range of potential values is big, then counting sort requires a lot of space (perhaps more than O (n) O(n) O(n)).
- Counting sort can only be used for arrays with integer elements because otherwise array of frequencies cannot be constructed.

3.10.6 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`

Output `arr = [2, 4, 5, 7, 8, 19, 23]`

3.11 Radix Sort

- Define 10 queues each representing a bucket for each digit from 0 to 9.
- Consider the least significant digit of each number in the list which is to be sorted.
- Insert each number into their respective queue based on the least significant digit.
- Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
- Repeat from step 3 based on the next least significant digit.
- Repeat from step 2 until all the numbers are grouped based on the most significant digit.

3.11.1 Pseudo code:

```
Radix-Sort (A, d)
for j = 1 to d do
int count [10] = {0};
for i = 0 to n do
count [key of(A[i]) in pass j] ++
for k = 1 to 10 do
count[k] = count[k] + count[k-1]
for i = n-1 downto 0 do
result [ count [key of(A[i])]] = A[j]
count [key of(A[i])]--
for i=0 to n do
A[i] = result[i]
end for(j)
end func
```

3.11.2 Python code:

```
def radixSort(array):

    max_element = max(array)

    place = 1
    while max_element // place > 0:
        countingSort (array, place)
        place *= 10
```

3.11.3 Time Complexity Analysis:

Worst Case Time Complexity **O(nd)**
Best Case Time Complexity **O(nd)**
Average Time Complexity **O(nd)**

3.11.4 Proof of Correctness:

- **Initialization** The array is trivially sorted on the last 00 digits.
- **Maintenance -** Let's assume that the array is sorted on the last i − 1 i−1 digits. After we sort on the ith h digit, the array will be sorted on the last ith digits. It is obvious that elements with different digit in the ith position are ordered accordingly; in the case of the same ith digit, we still get a correct order, because we're using a stable sort and the elements were already sorted on the last i − 1 i−1 digits.
- **Termination -** The loop terminates when i = d + 1i=d+1. Since the invariant holds, we have the numbers sorted on dd digits.

3.11.5 Strength:

- Sorting items that can be ordered based on their component digits or letters, such as integers, words.
- The grouping into buckets does not involve any comparisons
- Space complexity of the radix sort is better than the counting sort.

3.11.6 Weaknesses:

- Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. ...
- The constant for Radix sort is greater compared to other sorting algorithms.
- It takes more space compared to Quicksort which is inplace sorting.

3.11.7 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`
Output `arr = [2, 4, 5, 7, 8, 19, 23]`

3.12 Bucket Sort

- Suppose, the input array is (any unsorted one)
- Insert elements into the buckets from the array. The elements are inserted according to the range of the bucket.
- The elements of each bucket are sorted using any of the stable sorting algorithms. Here, we have used quicksort.
- The elements from each bucket are gathered.

3.12.1 Pseudo code:

```
BucketSort(A)
 n = A. length
Let B [0, ⋯, n − 1] be a new array
 for i = 0 to n - 1
B[i] ← 0
for i = 1 to n
B[bnA[i]c] ← A[i]
for i = 0 to n−1 sort list
B[i] using insertion sort
concatenate the lists B [0], B [1], ⋯, B [n − 1]
```

```
 return B
```

3.12.2 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
```

3.12.3 Time Complexity Analysis:

Worst Case Time Complexity **O(n^2)**
Best Case Time Complexity **O(n)**
Average Time Complexity **O (n + k)**

3.12.4 Proof of Correctness:

- Consider two elements A[i], A [j]
- Assume without loss of generality that A[i] ≤ A[j]
- Then nA[i] ≤ nA[j]
- A[i] belongs to the same bucket as A[j] or to a bucket with a lower index than that of A[j]
- If A[i], A[j] belong to the same bucket:
- sorting puts them in the proper order
- If A[i], A[j] are put in different buckets:
- concatenation of the lists puts them in the proper order

3.12.5 Strength:

- It is quicker to run than a bubble sort.
- Bucket sort allows each bucket to be processed independently. As a result, you'll frequently need to sort much smaller arrays as a secondary step after sorting the main array.
- Being able to be used as an external sorting algorithm. If you need to sort a list that is too large to fit in memory, you may stream it through RAM, split the contents into buckets saved in external files, and then sort each file separately in RAM.

3.12.6 Weaknesses:

- Buckets are distributed incorrectly; you may wind up spending a lot of extra effort for no or very little gain.

- Can’t apply it to all data types since a suitable bucketing technique is required. Bucket sort’s efficiency is dependent on the distribution of the input values, thus it’s not worth it if your data are closely grouped.
- Performance is determined by the number of buckets used, which may need some additional performance adjustment when compared to other algorithms.

3.12.7 Dry Run of python code:

Input `arr = [19, 5, 23, 8, 2, 4, 7]`

Output `arr = [2, 4, 5, 7, 8, 19, 23]`

4. UI

Final UI (User interface) made for this project on python language is explained below with the numbering of GUI (Graphical User Interface) components explanation:

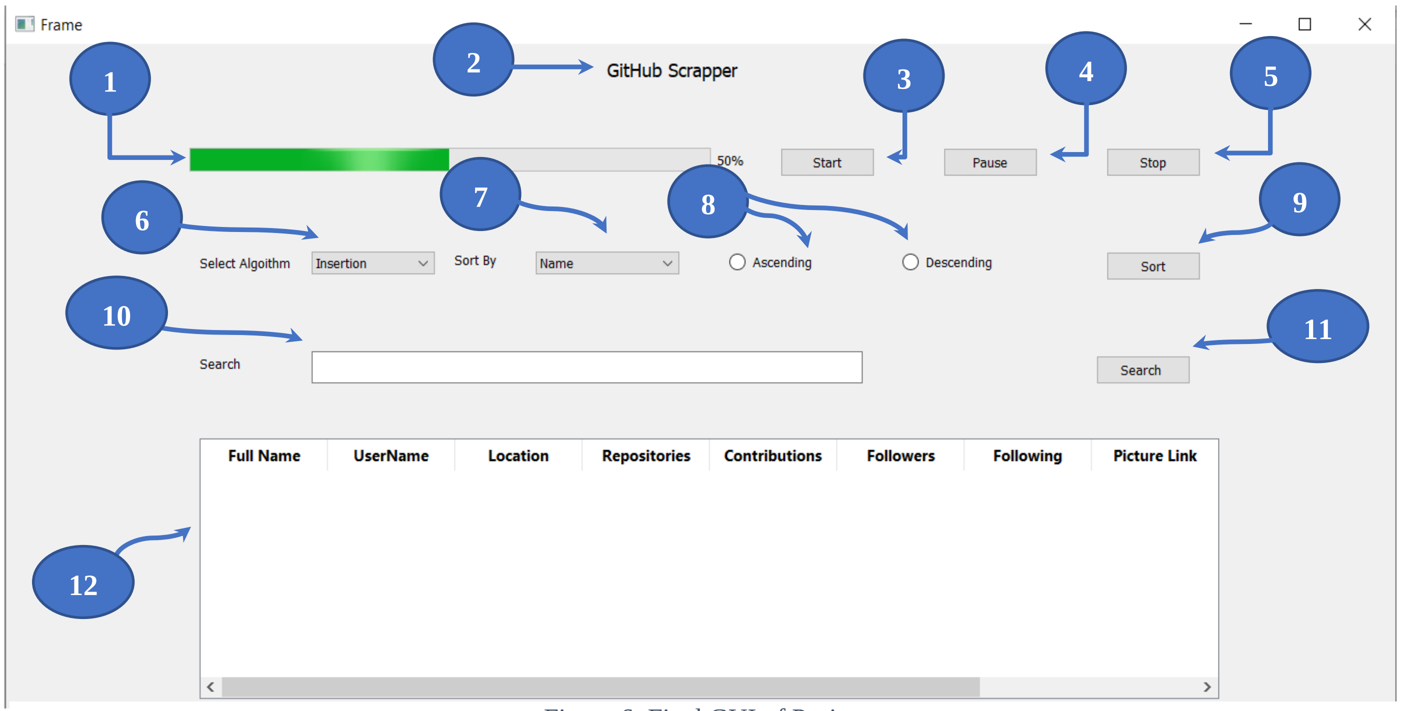


Figure 6: Final GUI of Project

Labeled components of figure 6 are described in the below table with GUI component name and function which the component has to perform in the project:

Sr.	**Component Name**	**Function**
1	Progress Bar	To show scrapping process in progress, and to show how much portion or percentage of data is scrapped.
2	Label	Just to show the label of project
3	Button	To start the process of scrapping
4	Button	To pause the process of scrapping
5	Button	To stop the process of scrapping
6	Combo Box	To show user option of sorting algorithms
7	Combo box	To show user options of columns to sort
8	Radio Button	To decide the user to sort data ascendingly or descending
9	Button	To start the process of sorting according to selected conditions (option 6,7 and 8)
10	Text Box	To get the string from user to search in scrapped data
11	Button	To start the process of searching
12	Data Grid View	To show scrapped data in the table.

5. Issues During Project

A project issue/problem is the difficulty that interrupt in the execution of project activities. Many issues occurred in completion of this project. Some of them are listed below:

5.1Scrapping Issues

Scrapping is one of the technical things in project requirement list. It was the great challenge for the team to grasp the concept and method of scrapping. Scrapping was new for the whole team. Scrapping issues faced by the team are some listed:

- Targeted website does not allow scrapping
- Understand the structure of web page to target required attributes
- Accessing elements without dives.
- Make sure not to block your pc for web site
- Slow/unstable load speed
- Double Scrapping required for project
- Need huge amount of data in less time

5.2Integration Issues

Integration is one of the tasks that occur at almost time of competition of project. But its one of the most difficult and necessary tasks that without project integration, other task has no importance as a project. So, some issues faced in this project are listed:

- Python new Language
- Python is not good in GUI
- Need to understand deep coding
- Shortage of time
- Need code for each and every event
- If need updating in GUI, whole cycle has to repeat
- All new tools to work

5.3Solutions
Where there is a problem there is way. Every solution exists only there where problem have shown its face. Problems that are appeared in this project are tried to over come by the following the given solutions:

- Use the pause function allow you to scrap some data from the web site that blocks you for scrapping and prevent your pc to block from web site.
- Practice for scrapping code by accessing multiple attributes like without div element accessed through child function.
- Make sure for stable internet connectivity
- Use multiple Laptops for scrapping data of this project to get more data in less time
- Make the attention to learn python language and practice of code on different platforms
- Use of PyQt designer to drag and drop the GUI components
- Leave the other work and make attention towards this project.

6. Project Team
This project is done in team or group with task division and management and task division. Information of the group members is described below.

6.1 Group Members
This project is done by the group of two students of CS department of UET Lahore.

- Muneeb ur Rehman (2019-CS-133)
- Muhammad Ali Murtaza (2020-CS-114)

6.2 Collaboration
For completion of this project, proper collaboration between the team members is made. Meetings and proper discussions are arranged and finally the project is concluded.

6.3 Task Division
In this whole journey of completion of this project, with collaboration tasks are divided among members of the team and divided in the following manner:

Sr.	Submission Task	Group Member
1	Project Proposal Report	2020-CS-114
2	UI (User Interface)	2020-CS-114
3	Algorithms Report	2019-CS-133
4	Scrapping Of Data	2019-CS-133 2020-CS-114
5	Integration of Project	2020-CS-114
6	Final Report	2020-CS-114

7. Conclusion
Following are the points for the conclusion of Final Report of scrapping project:

7.1 Final Application
This is the Final Scrapping Application in Python that is ready for scrapping and all above mentioned sorting features. It can scrap Profiles data up to any level. And Excel file or it self the application can be used to present the data.

7.2 In-progress improvements
Many features can be listed in this project for improvement

- More sorting algorithms ca be added for time efficient sorting.
- Some scrapping updates like fasting the process of scrapping, scrapping more data.
- Updated and better gui (Graphical user interface).
- Calculating estimated time calculating before for sorting for every algorithm working.
- Another progress bar for sorting data.
- Multi-level sorting

These are some updates that are in mind for advance version of this scrapping project.