

**YouTube Video Data Analysis**

**Submitted By:**

**Abdul Ahad (2020-CS-5)**

**Musawir Ahmed (2020-CS-46)**

**Submitted To:**

**Dr. Samyan Qayyum Wahla**

**For Fulfillment of**

**CS-262 Data Structures & Algorithms**

**Department of Computer Science**

**Table of Contents**

1. Project Overview
   * 1. Description
     2. Motivation
     3. Target Audience
     4. Business Need
2. Scraped Data
   * 1. Attributes
     2. Sample Scraping Source
3. Algorithms
   * 1. Sorting Algorithms
     2. Searching Algorithms
4. UI
   * 1. Initial GUI Design
     2. GUI Components
     3. Implemented GUI
5. Integration
   * 1. Problems and their Solutions
     2. Overview of Working
     3. Features After integration
     4. Missing Features
6. Collaboration

**Project Overview**

**Description**

We are making a retrieval system that can be used to process the raw data scraped off from YouTube to convert it into processed information. We will search keywords that will first be gathered from an API. A list will be populated first with such keywords and using those keywords, YouTube will be prompted for related videos. All the videos that are then shown on that webpage as a search result will then each be opened separately from where we will scrape the required data. We are aiming to scrape data of at least 1 million videos which is predicted to take several days. The user will be allowed to pause the scraping process and view the intermediate results of the scraping. The user can also resume the scrapping process. This data will then be saved in a CSV file which will be appended and shown in the application window in an appropriate tabular form. The user can also print the data scrapped. In this application, the user will be allowed to sort selected columns using algorithms of their own choice. The time taken to sort the results will also be given to the user. Moreover, the user can also sort multiple columns at the same time. Furthermore, the user is allowed to search the table for specific data items. The columns can also be sorted on multiple levels i.e., using Keywords like AND OR NOT to improve the search result accuracy. Advanced filters such as contains, starts with, ends with can be used to enhance the search results. A progress bar will be displayed to show the number of videos that have been scrapped.

**Motivation**

First and foremost, reason for making this specific retrieval system is to make things easier for content creators; In an effort to support the people who are clueless on how to grow their channels. Also, to deploy the sorting algorithms in a real-life example to observe the effect of sorting algorithm time complexities. Moreover, on a personal level it shall act as a stepping stone towards gaining experience and insight of the complexities underlying such systems that we take for granted and one day create an unparalleled and revolutionising DBMS software.

**Target Audience**

The people browsing YouTube for entertainment will be the target audience as the content creators will use the trends seen from the sorted data and utilise it to produce appropriate content.

The new content creators’ channel growth will stay as it already is if this project is not deployed officially: stunted and constant. This project aims to boost the growth and provide content ideas for YouTubers so that they may grow their channel. Moreover the project can be further improved in future to scrape data not only from YouTube but from any website the user wants which will be phenomenal and profitable if deployed officially as a Commercial Software.

**Scraped Data**

We scraped data from YouTube using hardcoded lists that were populated using the popular keywords and some keywords returned by API. The keywords are first searched and all the links related to the keyword entered are then scraped. Each link is then accessed where data of each video is scraped.

**Attributes:**

* **Video Name (String Type)**

This is going to be the name of the video whose data is being scrapped suggesting the type of content in video.

* **Channel Name (String Type)**

This is the name of the channel corresponding to the video that has been scraped.

* **Subscribers (Integers)**

This attribute will hold the number of subscribers that correspond to that channel

* **Views (Integer)**

These are the number of views.

* **Likes (Integer)**

The number of likes a video gets will be scraped and stored in this.

* **Dislikes (Integer)**

The number of dislikes a video gets will be saved in this attribute.

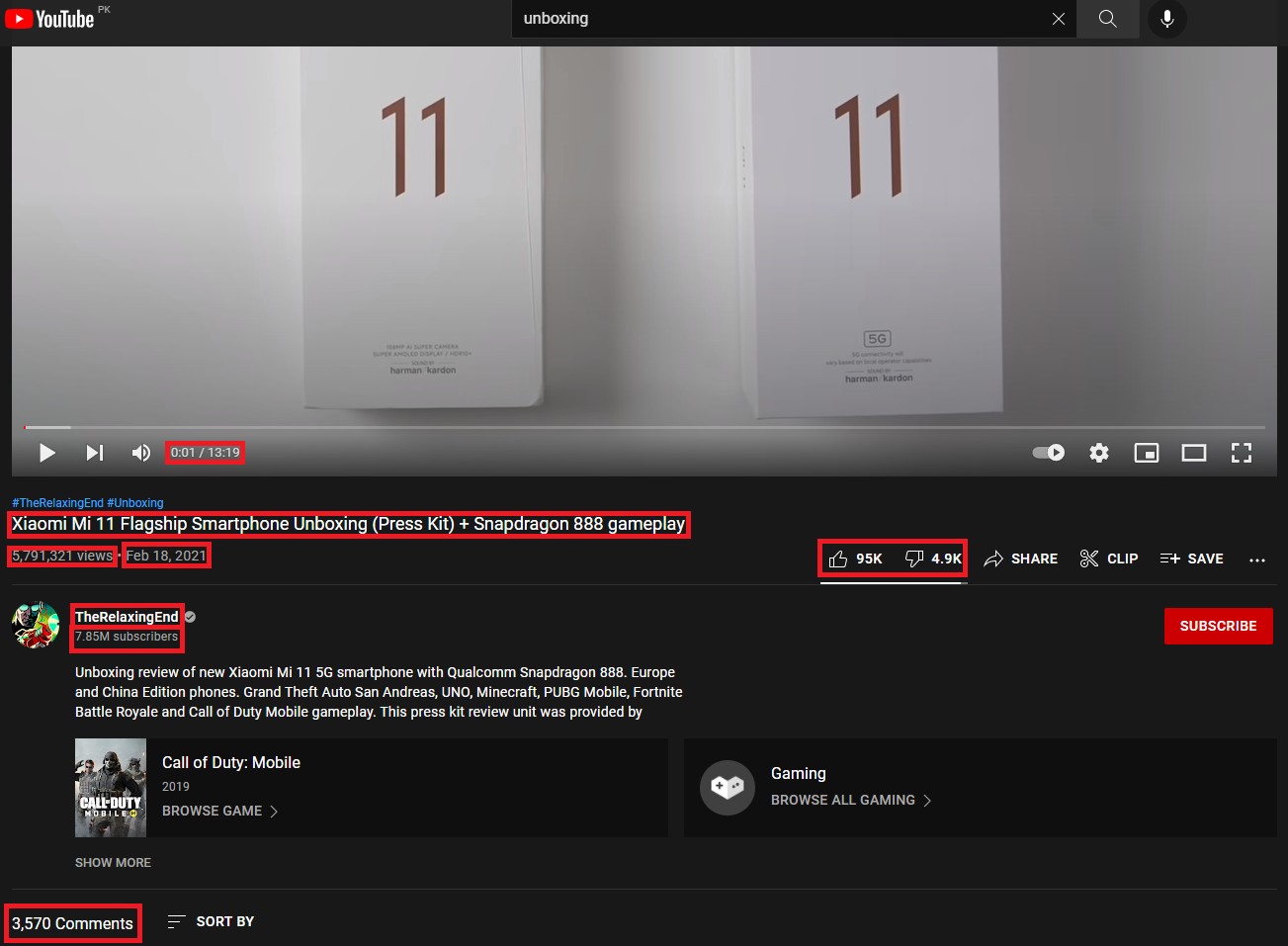
* **Comments (Integer)**

The number of comments underneath a video

* **Publishing Date (Date)**

The date the video was published for public view

**Sample scraping source**



**Sorting Algorithms**

**Insertion Sort**

The algorithm starts from the 2nd index skipping the first element as it is considered sorted already. A comparison is made with the previous element and if the previous element is greater than the selected element, it is not swapped; otherwise, the element that is selected gets overwritten by the preceding element till the position it is supposed to be in is found.

**Pseudocode**

Insertion\_Sort(A,SortType)

for i = 2 to A.length

key = i

j = i-1

if SortType == "Ascending"

while j>-1 and A[j]>A[key]

A[j+1]=A[j]

j = j - 1

else

while j>-1 and A[j]<A[key]

A[j+1]=A[j]

j = j - 1

A[j+1] = A[key]

**Code in Python**

def insertion\_Sort(A,SortType):

for i in range(len(A)):

key = A[i]

j = i-1

while j>-1 and A[j]>=key:

A[j+1]=A[j]

j-=1

A[j+1] = key

if SortType == "Ascending":

return A

return list(reversed(A))

**Time Complexity**

def insertion\_Sort(A,SortType):

    for i in range(1,len(A)): n

        key = i n-2

        j = i-1 n-2

        while j>-1 and A[j]>A[key]:

             A[j+1]=A[j]

             j-=1

        A[j+1] = A[key]

if SortType == “Ascending”: 1

return A 1

return list(reversed(A)) 1

**Proof of correctness**

Initialization

Before the first iteration of the loop, we can see that the loop that starts at the 2nd index that is 1, we see that the subarray formed A[0…i-1] is already sorted as it consists of just one element A[0] so it’ll be considered sorted and so the loop invariant holds before the loop starts

Maintenance

The loop iterates and increments the counter i and continues to sort the array. The inner loop moves each element one step back till the element is found to be either lesser or equal to the element to be inserted or the end of the array is reached. Now the subarray A[0…i] is considered sorted and subarray A[0…i-1] is sorted when the element has been inserted and counter incremented. So the loop invariant holds while-loop continues to iterate.

Termination

The end of the for loop is achieved when the counter becomes greater than the upper limit which is equal to n. The counter i’s value at the end is one up than the range so i = n+1 and so the subarray A[0…i-1] becomes A[n] which is basically the original array only with sorted elements so the loop invariant holds even after loop termination

**Strengths**

1. Simple
2. Efficient for a small input
3. Memory requirement is minimized since it sorts the elements in place

**Weaknesses**

1. There are better sorting algorithms that sort in O(n) time
2. If the input is large, the time taken to sort increases exponentially
3. Can only be used for small inputs so it isn't efficient for larger inputs

**Dry Run**

Original Array

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 3 | 4 | 1 | 7 | 5 |

1st iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 3 | 4 | 1 | 7 | 5 |

2nd iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 3 | 4 | 1 | 7 | 5 |

3rd iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 9 | 10 | 4 | 1 | 7 | 5 |

4th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 4 | 9 | 10 | 1 | 7 | 5 |

5th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 9 | 10 | 7 | 5 |

6th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 7 | 9 | 10 | 5 |

7th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

**Merge Sort**

The algorithm keeps breaking down the array in left and right half till only one element remains in an array. The elements in the left and right arrays are compared and then they are sorted in the main array by copying the elements from the two arrays formed while breaking the main array.

**Pseudocode**

Merge\_Sort(A,left,right,SortType)

if right > left

mid = floor((left + right)/2)

Merge\_Sort(A,left,mid)

Merge\_Sort(A,mid+1,right)

Merge(A,left,mid,right,SortType)

Merge(A,p,q,r,SortType):

n1 = q-p+1

n2 = r-q

Make L array //left array of size n1

Make R array //right array of size n2

for i = 1 to n1

L[i] = A[p+i]

for j = 1 to n2

R[j]=A[q+1+j]

if SortType == “Ascending”

L[n1] = infinity

R[n2] = infinity

else

L[n1] = -infinity

R[n2] = -infinity

i = 0

j = 0

for k = p to r

if SortType == "Ascending"

if L[i]<=R[j]

A[k] = L[i]

i = i +1

else:

A[k] = R[j]

j =j +1

else:

if L[i]>=R[j]

A[k] = L[i]

i =i+1

else:

A[k] = R[j]

j=j+1

**Code in Python**

def merge\_Sort(A,left,right,SortType):

if right > left:

mid = (left + right)//2

merge\_Sort(A,left,mid,SortType)

merge\_Sort(A,mid+1,right,SortType)

Merge(A,left,mid,right)

if SortType=="Ascending":

return A

return list(reversed(A))

def Merge(A,p,q,r):

n1 = q-p+1

n2 = r-q

L = []

R = []

for i in range(n1):

L.append(A[p+i])

for j in range(n2):

R.append(A[q+1+j])

L.append(math.inf)

R.append(math.inf)

i = 0

j = 0

for k in range(p,r+1):

if L[i]<=R[j]:

A[k] = L[i]

i+=1

else:

A[k] = R[j]

j+=1

**Time Complexity**

def Merge(A,p,q,r,SortType):

    n1 = q-p+1 1

    n2 = r-q 1

    L= [] 1

    R = [] 1

    for i = 1 to n1 (for first division)

        L.append(A[p+i])

    for j in range(n2):

        R.append(A[q+1+j])

    L.append(-math.inf) 1

    R.append(-math.inf) 1

    i = 0  1

    j = 0 1

    for k in range(p,r+1): n+1

            if L[i]>=R[j]: n

                A[k] = L[i] n

                i+=1 n

            else: n

                A[k] = R[j] n

                j+=1 n

The merge function takes O(n) time to combine. The division takes O(1) time. The problem is being solved using divide and conquer so each subproblem is of size n/2 and there are 2 subproblems. Each level of division takes c(n) time where c is constant. So, there being lg n+1 total levels, the total time taken would be cn(lg n+1) which is cnlgn + cn = O(nlgn)

**Proof Correctness**

Initialization

Before the loop begins the subarray A[p…k-1] is empty. This empty subarray contains 0 smallest elements of L and R as the counters of both arrays i and j are equal to 1. Before the loop begins both L[i] and R[j] are the smallest elements because the pointer is at the base of the array. These elements have not yet been placed into array A.

Maintenance

Let L[i]<=R[j]. Then L[i] is the smallest element not yet copied back into A. This temporarily disturbs the equilibrium which is soon re-established when this smallest element is copied into the array A. The array A now contains k-p+1 smallest elements. The counter k and i are incremented which bring the equilibrium back. It works in the same way if L[i]> R[j] to maintain the loop invariant.

Termination

At the end of the for-loop the counter k = r+1. If we substitute the value of k in the subarray A[p…k-1] we get A[p…r] which contains the r-p+1 smallest elements of L and R only in sorted order. Here the sentinels aren’t copied back.

**Strengths**

1. It performs extremely well on large inputs
2. In worst case it takes O(nlogn) time

**Weakness**

1. Requires more memory space as the merge function creates 2 more arrays everytime it is called of size n/2
2. Less efficient at smaller inputs
3. There are sorting algorithms that sort in O(n) time

**Dry Run**

1st recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 3 | 4 | 1 | 7 | 5 |

2nd recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 3 | 4 | 1 | 7 | 5 |

3rd recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 4 | 9 | 10 | 1 | 7 | 5 |

4th recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 4 | 9 | 10 | 1 | 7 | 5 |

5th recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 4 | 9 | 10 | 1 | 5 | 7 |

6th recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

**Strand Sort**

The first element of the array is copied in a newly made subarray. The array is then traversed for elements larger than the previous element entered in the sub array till the end of the main array while popping off the elements added in the sub array. The process repeats for the next element till the main array is empty. These elements are then copied in the final array which is initially empty. The method of copying is the one used in merge sort where two counters are used and elements compared from two arrays.

**Pseudocode**

Merge\_strand(newarr,outarr,SortType):

make a list returnarr to return

while len(newarr)>0 and len(outarr)>0

if SortType == "Ascending"

if newarr[0]>=outarr[0]

returnarr.append(outarr[0])

outarr.pop(0)

else:

returnarr.append(newarr[0])

newarr.pop(0)

else:

if newarr[0]<=outarr[0]

returnarr.append(outarr[0])

outarr.pop(0)

else:

returnarr.append(newarr[0])

newarr.pop(0)

returnarr = returnarr + newarr

returnarr = returnarr + outarr

return returnarr

Strand\_Sort(inArr,SortType)

Make an array outputarr //this array will be output in sorted form

while inArr.length!=0

newarr = []

main\_count = 0

newarr.append(inArr[0])

prev\_element = newarr[0]

inArr.pop(0)

if SortType == "Ascending"

while main\_count!=inArr.length

if inArr[main\_count]>prev\_element

prev\_element = inArr[main\_count]

newarr.append(prev\_element)

inArr.pop(main\_count)

main\_count = main\_count-1

main\_count = main\_count+1

else:

while main\_count!=inArr.length

if inArr[main\_count]<prev\_element

prev\_element = inArr[main\_count]

newarr.append(prev\_element)

inArr.pop(main\_count)

main\_count = main\_count-1

main\_count = main\_count+1

outputarr =Merge\_strand(newarr,outputarr,SortType)

return outputarr

**Code in Python**

def merge\_strand(newarr,outarr,SortType):

returnarr = []

while len(newarr)>0 and len(outarr)>0:

if SortType == "Ascending":

if newarr[0]>=outarr[0]:

returnarr.append(outarr[0])

outarr.pop(0)

else:

returnarr.append(newarr[0])

newarr.pop(0)

else:

if newarr[0]<=outarr[0]:

returnarr.append(outarr[0])

outarr.pop(0)

else:

returnarr.append(newarr[0])

newarr.pop(0)

returnarr += newarr

returnarr += outarr

return returnarr

def strand\_Sort(inArr,SortType):

outputarr = []

while len(inArr)!=0:

newarr = []

main\_count = 0

newarr.append(inArr[0])

prev\_element = newarr[0]

inArr.pop(0)

if SortType == "Ascending":

while main\_count!=len(inArr):

if inArr[main\_count]>prev\_element:

prev\_element = inArr[main\_count]

newarr.append(prev\_element)

inArr.pop(main\_count)

main\_count-=1

main\_count+=1

else:

while main\_count!=len(inArr):

print(inArr[main\_count],prev\_element,"\t",len(inArr),main\_count)

print(newarr,outputarr,inArr)

print("")

if inArr[main\_count]<prev\_element:

prev\_element = inArr[main\_count]

newarr.append(prev\_element)

inArr.pop(main\_count)

main\_count-=1

main\_count+=1

outputarr = merge\_strand(newarr,outputarr,SortType)

return outputarr

**Time Complexity**

The Merge function takes a maximum of time O(n) time in worst case

def strand\_Sort(inArr,SortType):

    outputarr = [] 1

    while len(inArr)!=0: n+1

        newarr = [] n-1

        main\_count = 0 n-1

        newarr.append(inArr[0]) n-1

        prev\_element = newarr[0] n-1

        inArr.pop(0) n-1

        if SortType == "Ascending": n-1

            while main\_count!=len(inArr): (n-1)(n+1)

                if inArr[main\_count]>prev\_element: n2

                    prev\_element = inArr[main\_count] n2

                    newarr.append(prev\_element) n2

                    inArr.pop(main\_count) n2

                    main\_count-=1 n2

                main\_count+=1 n2

        else: n-1

            while main\_count!=len(inArr): n2-1

                if inArr[main\_count]<prev\_element: n2

                    prev\_element = inArr[main\_count] n2

                    newarr.append(prev\_element) n2

                    inArr.pop(main\_count) n2

                    main\_count-=1 n2

                main\_count+=1 n2

        outputarr = merge\_strand(newarr,outputarr,SortType) n(n-1)

    return outputarr n-1

**Strengths**

1. Sorts in O(n) in best case scenario

**Weakness**

1. Sorts in O(n\*n) in worst case scenario
2. It requires more space in memory as it is not inplace sorting

**Dry Run**

Input

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 3 | 4 | 1 | 7 | 5 |

1st iteration

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 3 | 1 | 7 | 5 |

2nd iteration

|  |  |  |
| --- | --- | --- |
| 3 | 1 | 5 |

3rd iteration

|  |
| --- |
| 1 |

4th iteration

|  |
| --- |
|  |

Output Array

|  |
| --- |
|  |

1st iteration

|  |  |
| --- | --- |
| 9 | 10 |

2nd iteration

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 7 | 9 | 10 |

3rd iteration

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 5 | 7 | 9 | 10 |

4th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

**Quick Sort**

The algorithm recursively selects a pivot random and puts all the elements smaller than it on the left side and greater elements on its right side without actually sorting them. Each element then gets placed in its position by a pivot as the problem gets divided till there is one element left in an array which is then considered sorted.

**Pseudocode**

Quick\_Sort(arr,low,high,SortType):

if low<high

pivot = partition(arr,low,high,SortType)

Quick\_Sort(arr,low,pivot-1)

Quick\_Sort(arr,pivot+1,high)

def partition(arr,low,high,SortType):

pivot = arr[high]

i = low - 1

for j in range(low,high):

if SortType=="Ascending":

if arr[j]<pivot:

i=i+1

swap arr[i] and arr[j]

else:

if arr[j]>pivot:

i=i+1

swap arr[i] and arr[j]

swap arr[high] and arr[i+1]

return i+1

**Code in Python**

def quick\_Sort(arr,low,high,SortType):

if low<high:

pivot = partition(arr,low,high)

quick\_Sort(arr,low,pivot-1,SortType)

quick\_Sort(arr,pivot+1,high,SortType)

if SortType == "Ascending":

return arr

return list(reversed(arr))

def partition(arr,low,high):

pivot = arr[high]

i = low - 1

for j in range(low,high):

if arr[j]<pivot:

i+=1

arr[i],arr[j] = arr[j], arr[i]

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

return i+1

**Time Complexity**

Each problem is divided into 2 subproblems each of size n/2 so there are lg n+1 levels. Each level takes cn so like merge sort the time complexity is O(nlgn) in the case the pivot chosen is a median which is best case. In the worst case it’d perform O(n2) which is when the pivot chosen is either the largest of all the elements or smallest and so one recursive call will take O(0) and in the same flow the other recursive call will take O(n2) as the whole input is passed down as a subproblem.

**Proof Correctness**

Initialization

Before the loop begins the partition function maintains 4 parts of the array. The first part of the array being A[r] = pivot and r is the last element of the array. The subarray A[low…i] where i = low-1 doesn’t have any element in it so all elements less than pivot are present is true and neither does the subarray A[i+1…j] as j = low and i+1 = low so all elements greater than pivot are present in this is true. The subarray A[j+1…r] is unrestricted meaning elements greater or less than may exist here is also true as all the elements are present here in this array of the original array.

Maintenance

When the value A[j] is greater than pivot then the value of j is simply incremented. The 2nd condition satisfies for j-1 as the A[i+1…j-1] now contains the element(s) greater than pivot. If A[j] < pivot then i is incremented and A[i] and A[j] swapped after which j is incremented. The 1st condition satisfies as A[low…i] now contains elements smaller than pivot.

Termination

At the end of the loop, the value of j becomes equal to r. The value of j is not further incremented so subarray A[low…i] now contains all elements smaller than pivot and A[i+1…j] contains elements greater than pivot.

**Strengths**

1. Considered to be the best sorting algorithm
2. Does not require additional space like merge sort. Sorts elements in place
3. Highly efficient with large input sizes

**Weakness**

1. In the worst case scenario, it performs like insertion sort’s average time O(n^2)
2. If the list is already sorted, it takes more time than any other sorting would take
3. Not stable

**Dry Run**

input

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 3 | 4 | 1 | 7 | 5 |

1st recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 3 | 1 | 5 | 9 | 7 | 10 |

2nd recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

3rd recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

4th recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

5th recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

6th recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

7th recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

8th recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

9th recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

10th recursive call

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

**Radix Sort**

The elements are first put in a bucket which is a visual representation of a virtually created array. The numbers are then sorted while keeping stability as FIFO(FIRST IN FIRST OUT) is used. The numbers are now rearranged in the main array with respect to the first digit. Then the process repeats till the maximum number of digits that are present which is equal to the digits of the largest number present.

**Pseudocode**

Radixcounting\_sort(passed\_array,divident)

orignal\_array=passed\_array.copy()

for x =1 to passed\_array.length

condition=FALSE

if(passed\_array[x]<0)

condition=True

passed\_array[x]=passed\_array[x]\*-1

passed\_array[x]=passed\_array[x]/divident

passed\_array[x]=int(passed\_array[x]%10)

if(condition==TRUE)

passed\_array[x]=passed\_array[x]\*-1

min\_vallue=min(passed\_array)

if(min\_vallue<0)

for x =1 to passed\_array.length

passed\_array[x]=passed\_array[x]-(min\_vallue)

max\_value=max(passed\_array)+1

make an array counting\_array

make an array resultant\_array

for x=1 to max\_value

counting\_array.append(0)

for x=1 to passed\_array.length

resultant\_array.append(0)

for x =1 to passed\_array.length

index=passed\_array[x]

counting\_array[index]=counting\_array[index]+1

for x =1 to counting\_array.length-1

counting\_array[x+1]=counting\_array[x+1]+counting\_array[x]

for x =passed\_array.length downto 1

index=passed\_array[x]

sorted\_index=counting\_array[index]

counting\_array[index]=counting\_array[index]-1

resultant\_array[x]=sorted\_index-1

for x = 1 to resultant\_array.length

index=resultant\_array[x]

passed\_array[index]=orignal\_array[x]

return passed\_array

Redix\_sort(passed\_array,SortType):

max\_value=max(passed\_array)

divident=1

while int(max\_value/divident) >0

passed\_array=Radixcounting\_sort(passed\_array,divident)

divident=divident\*10

if SortType=="Ascending":

return passed\_array

return list(reversed(passed\_array))

**Code in Python**

def radixcounting\_sort(passed\_array,divident):

orignal\_array=passed\_array.copy()

for x in range(0,len(passed\_array)):

condition=False

if(passed\_array[x]<0):

condition=True

passed\_array[x]=passed\_array[x]\*-1

passed\_array[x]=passed\_array[x]/divident

passed\_array[x]=int(passed\_array[x]%10)

if(condition==True):

passed\_array[x]=passed\_array[x]\*-1

min\_vallue=min(passed\_array)

if(min\_vallue<0):

for x in range(0,len(passed\_array)):

passed\_array[x]=passed\_array[x]-(min\_vallue)

max\_value=max(passed\_array)+1

counting\_array=[]

resultant\_array=[]

for x in range (0,max\_value):

counting\_array.append(0)

for x in range(0,len(passed\_array)):

resultant\_array.append(0)

for x in range(0,len(passed\_array)):

index=passed\_array[x]

counting\_array[index]=counting\_array[index]+1

for x in range(0,len(counting\_array)-1):

counting\_array[x+1]=counting\_array[x+1]+counting\_array[x]

for x in range(len(passed\_array)-1,-1,-1):

index=passed\_array[x]

sorted\_index=counting\_array[index]

counting\_array[index]=counting\_array[index]-1

resultant\_array[x]=sorted\_index-1

for x in range(0,len(resultant\_array)):

index=resultant\_array[x]

passed\_array[index]=orignal\_array[x]

return passed\_array

def redix\_sort(passed\_array,SortType):

max\_value=max(passed\_array)

divident=1

while int(max\_value/divident) >0:

passed\_array=radixcounting\_sort(passed\_array,divident)

divident=divident\*10

if SortType=="Ascending":

return passed\_array

return list(reversed(passed\_array))

**Time Complexity**

Each for loop in the counting sort utilized by radix sort function runs a maximum of O(n). The radix sort takes O(d × (n)) , d = where k is the largest number in the array and b is the base of the number being represented. We make the base of the number b as n to decrease the value of d to 1 and give time complexity of O(n)

**Strengths**

1. Sorts in in O(n) time
2. If the range of numbers is small then it becomes the most efficient
3. It is a stable sort algorithm

**Weakness**

1. For different data type it needs to be modified as it relies on the digits and letters
2. It takes up more space in memory as it creates more arrays
3. For float point values, bucket sort is better

**Dry Run**

input

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 90 | 100 | 40 | 3 | 10 | 70 | 50 |

1st iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 90 | 100 | 40 | 10 | 70 | 50 | 3 |

2nd iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 100 | 3 | 10 | 40 | 50 | 70 | 90 |

3rd iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 10 | 40 | 50 | 70 | 90 | 100 |

**.**

**Pigeonhole Sort**

Find the maximum value and the minimum value from the array. Find the Range of the array elements and form a new array of size of range. The array is then traversed from the start till end. Each element is then placed in this new array at the index calculated by subtracting minimum value from the element. At the end, all the elements are then copied back to the original array keeping sort stable.

**Pseudocode**

Pigeonhole\_Sort(A,SortType):

maxelement = max(A)

minelement = min(A)

arr\_range = maxelement-minelement+1

Make pigeonhole array

for k =1 to arr\_range

pigeonhole.append([]\*arr\_range)//add array within the array

for i =1 to A.length

piglist = pigeonhole[A[i]-minelement]

piglist.append(A[i])

pigeonhole[A[i]-minelement] = piglist

arraycount = 0

for j = 1 to pigeonhole.length

for c =1 to pigeonhole[j].length:

piglist = pigeonhole[j]

if piglist:

A[arraycount] = piglist[c]

arraycount+=1

if SortType == "Ascending":

return A

return list(reversed(A))

**Code in Python**

def pigeonhole\_Sort(A,SortType):

maxelement = max(A)

minelement = min(A)

arr\_range = maxelement-minelement+1

pigeonhole = []

for k in range(arr\_range):

pigeonhole.append([]\*arr\_range)

for i in range(len(A)):

piglist = pigeonhole[A[i]-minelement]

piglist.append(A[i])

pigeonhole[A[i]-minelement] = piglist

arraycount = 0

print(pigeonhole)

for j in range(len(pigeonhole)):

for c in range(len(pigeonhole[j])):

piglist = pigeonhole[j]

if piglist:

A[arraycount] = piglist[c]

arraycount+=1

if SortType == "Ascending":

return A

return list(reversed(A))

**Time Complexity**

Pigeonhole\_Sort(A,SortType):

    maxelement = max(A) 1

    minelement = min(A) 1

    arr\_range = maxelement-minelement+1 1

    Make pigeonhole array 1

    for k =1 to arr\_range 1

        pigeonhole.append([]\*arr\_range) 1

    for i =1 to A.length n+1

        piglist = pigeonhole[A[i]-minelement] n

        piglist.append(A[i]) n

        pigeonhole[A[i]-minelement] = piglist n

    arraycount = 0  1

    for j = 1 to pigeonhole.length n+1

        for c =1 to pigeonhole[j].length: n(N)

            piglist = pigeonhole[j] nN - 1

            if piglist: nN - 1

                A[arraycount] = piglist[c] nN - 1

                arraycount+=1 nN - 1

    return A 1

The time complexity is O(n+N) where N is the range of numbers in the array.

**Strengths**

1. Best algorithm when the range of elements and the number of elements is approximately the same
2. Stable sort
3. Does not compares elements for sorting

**Weakness**

1. There are other algorithms that perform better in linear time for instance counting sort or bucket sort
2. The requirements are rarely met and so it is not used that much.

**Dry Run**

input

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 4 | 3 | 1 | 7 | 5 |

1st iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 10 | 4 | 3 | 1 | 7 | 5 |

2nd iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 3 | 1 | 7 | 5 |

3rd iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 3 | 1 | 7 | 5 |

4th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 1 | 7 | 5 |

5th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 7 | 5 |

6th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 5 |

7th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

**Cocktail Sort**

The algorithm traverses the array and finds the maximum value and places it at the end of the array. Then it traverses from the end of the array ignoring the very last index at which the element was replaced and finds the minimum value to place it at the first index. This process then repeats as the algorithm finds maximum when moving to right and minimum when moving to left to place them at the ends of the array till the array is completely sorted.

**Pseudocode**

Cocktail\_Sort(A,SortType)

minidx = 0

maxidx = A.length-1

for k =1 to len(A)

###

if k MOD 2 == 0:

maxval = -infinity

for i = minidx to maxidx+1

if A[i]>maxval:

maxval = A[i]

maxval\_idx = i

A[maxval\_idx]= A[maxidx]

A[maxidx] = maxval

maxidx=maxidx - 1

###

else:

minval = infinity

for i =maxidx downto minidx-1

if A[i]<minval:

minval = A[i]

minval\_idx = i

A[minval\_idx] = A[minidx]

A[minidx] = minval

minval = minval + 1

if SortType == "Ascending":

return A

return list(reversed(A))

**Code in Python**

def cocktail\_Sort(A,SortType):

minidx = 0

maxidx = len(A)-1

for k in range(len(A)):

###

if k%2==0:

maxval = -math.inf

for i in range(minidx,maxidx+1):

if A[i]>maxval:

maxval = A[i]

maxval\_idx = i

A[maxval\_idx],A[maxidx] = A[maxidx],maxval

maxidx -= 1

###

else:

minval = math.inf

for i in range(maxidx,minidx-1,-1):

if A[i]<minval:

minval = A[i]

minval\_idx = i

A[minval\_idx],A[minidx] = A[minidx],minval

minval += 1

if SortType == "Ascending":

return A

return list(reversed(A))

**Time Complexity**

The time complexity of cocktail sort is O(n2) but it is better than bubble sort. It is a little less than two times faster than bubble sort. It takes n+1 time for the outer loop. Either of the inner loops takes n time for the first outer loop iterations after which either of the loops runs n-k times and so which is approximately O(n2) but better than bubble sort.

**Strengths**

1. Faster than bubble sort
2. It does not requires more memory allocation as it does inplace sorting
3. Sorts in O(n) time

**Weakness**

1. Not efficient than quick sort
2. Worst case and average case time complexity can reach O(n^2)

**Dry Run**

input

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 10 | 4 | 3 | 1 | 7 | 5 |

1st iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 5 | 4 | 3 | 1 | 7 | 10 |

2nd iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 5 | 4 | 3 | 9 | 7 | 10 |

3rd iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 5 | 4 | 3 | 7 | 9 | 10 |

4th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 5 | 4 | 3 | 7 | 9 | 10 |

5th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 5 | 4 | 3 | 7 | 9 | 10 |

6th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 5 | 4 | 3 | 7 | 9 | 10 |

7th iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 9 | 10 |

**Selection Sort**

The algorithm executes by finding the smallest element in the array and then placing it at the start of the array. Then it traverses along the array from the next index instead of starting from the previous index at which the element selected previously was swapped at.

**Pseudocode**

selection\_sort(Array,sorttype):

expression=None

if(sorttype=="Ascending"):

expression="Array[index]>Array[y]"

else:

expression="Array[index]<Array[y]"

for x =0 to len(Array):

index=x

for y=x+1 to len(Array):

if(eval(expression)):

index=y

temp=Array[index]

Array[index]=Array[x]

Array[x]=temp

return Array

**Code in Python**

def selection\_sort(Array,sorttype):

expression=None

if(sorttype=="Ascending"):

expression="Array[index]>Array[y]"

else:

expression="Array[index]<Array[y]"

for x in range(0,len(Array)):

index=x

for y in range(x+1,len(Array)):

if(eval(expression)):

index=y

temp=Array[index]

Array[index]=Array[x]

Array[x]=temp

return Array

**Time Complexity**

def selection\_sort(Array,sorttype):

expression=None\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

if(sorttype=="Ascending"):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

expression="Array[index]>Array[y]"

else:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

expression="Array[index]<Array[y]"

for x in range(0,len(Array)):\_\_\_\_\_\_\_\_\_\_\_\_\_\_n+1

index=x\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n

for y in range(x+1,len(Array)):\_\_\_\_\_\_\_\_\_\_n(n+1)

if(eval(expression)):\_\_\_\_\_\_n(n)

index=y

temp=Array[index]\_\_\_\_\_\_\_\_n

Array[index]=Array[x]\_\_\_\_\_n

Array[x]=temp\_\_\_\_\_\_\_\_\_\_\_n

return Array \_\_\_\_\_\_\_\_\_1

Time Complexity Will be O(n^2)

**Proof Correctness**

**Initialisation**

At the start of the loop before the first iteration we can see that if we form a subarray A[1…(i-1)], it will contain elements that are already sorted. In this case the subarray initially would be empty and so it will be sorted.

**Maintenance**

After each iteration, the outer for loop maintains the invariant. The subarray A[1…(i-1)] contains elements that were selected by the inner for loop and placed at their appropriate positions till i-1 index. The loop re-iterates preserving the invariant while incrementing i.

**Termination**

When the outer for loop ends, the for-loop header is executed once more than the body instructions. The counter i becomes equal to n+1 hence it terminates the loop. The subarray A[1…(i-1)] still contains the elements selected and sorted, substituting we get A[n] which is equal to the elements of the original array only sorted.

**Strengths**

1. It performs well on small lists.
2. No additional data structure is required as it is in place sorting algorithm.
3. Selection sort can be changed to stable sort or unstable sort based on the condition given in if.

**Weakness**

1. It has the time complexity of O(n^2)
2. poor performance on large number if list

**Dry Run**

Original Array

|  |  |  |  |
| --- | --- | --- | --- |
| 7 | 2 | 6 | 3 |

Iteration 0

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 7 | 6 | 3 |

Iteration 1

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 6 | 7 |

Iteration 2

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 6 | 7 |

Iteration 3

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 6 | 7 |

**Bubble Sort**

Each element is swapped with its adjacent element till it reaches the right position. It is named bubble sort because if an array represented vertically, each element from the bottom of the array rises to the top like a bubble would from under the water

**Pseudocode**

Buble\_sort(Array,sorttype):

expression=None

if(sorttype=="Ascending"):

expression="Array[y]<Array[x]"

else:

expression="Array[y]>Array[x]"

for x =0 to len(Array):

for y =x+1 to len(Array):

if(eval(expression)):

temp=Array[y]

Array[y]=Array[x]

Array[x]=temp

return Array

**Code in Python**

def Bubble\_sort(Array,sorttype):

expression=None

if(sorttype=="Ascending"):

expression="Array[y]<Array[x]"

else:

expression="Array[y]>Array[x]"

for x in range(0,len(Array)):

for y in range(x+1,len(Array)):

if(eval(expression)):

temp=Array[y]

Array[y]=Array[x]

Array[x]=temp

return Array

**Time Complexity**

def Bubble\_sort(Array,sorttype):

expression=None\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

if(sorttype=="Ascending"):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

expression="Array[y]<Array[x]"

else:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

expression="Array[y]>Array[x]"

for x in range(0,len(Array)):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n+1

for y in range(x+1,len(Array)):\_\_\_\_\_\_n(n+1)

if(eval(expression)):\_\_\_\_\_\_\_\_\_n(n)

temp=Array[y]\_\_\_\_\_\_\_\_\_n(n)

Array[y]=Array[x]\_\_\_\_\_\_\_\_\_n(n)

Array[x]=temp\_\_\_\_\_\_\_\_\_n(n)

return Array\_\_\_\_\_\_\_\_\_1

Time Complexity Will Be O(n^2)

**Strengths**

1. Uses in place sorting techniques thus additional data structure is not required.
2. Efficient when used for a small number of lists.
3. Suitable for Academic teaching as it has easy implementation.We also learned this algorithm as our first sorting algorithm

**Weakness**

1. It has the time complexity of O(n^2)
2. Not suitable for sorting large number of list.
3. (Less efficient)

**Dry Run**

Original Array

|  |  |  |  |
| --- | --- | --- | --- |
| 7 | 2 | 6 | 3 |

Iteration 0

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 7 | 6 | 3 |

Iteration 1

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 7 | 6 |

Iteration 2

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 6 | 7 |

Iteration 3

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 6 | 7 |

**Counting Sort**

An array is formed with the size of the largest element in the array. Then the elements from the original array are counted the number of times they are being repeated. This count is then placed at the index corresponding to the element counted in the new array. Then this count array is updated by taking the cumulative sum to give the index positions. While keeping the sort stable, the elements from the main array are now copied into the third array of the same size as the original using the count from the 2nd array that was created.

**Pseudocode**

counting\_sort(passed\_array,sorttype):

min\_vallue=min(passed\_array)#Find Minimum element in list

if(min\_vallue<0):

for x =0 to len(passed\_array):

passed\_array[x]=passed\_array[x]-(min\_vallue)

max\_value=max(passed\_array)+1 #Find Max element in list ///////and add 1

counting\_array[max\_value]#Create list of length max\_value

resultant\_array[len(passed\_array)]# create array of length ////////passed array

for x =0 to len(passed\_array):

index=passed\_array[x]

counting\_array[index]=counting\_array[index]+1

for x =0 to len(counting\_array)-1):

counting\_array[x+1]=counting\_array[x+1]+counting\_array[x]

for x =len(passed\_array)-1 to -1):

index=passed\_array[x]

sorted\_index=counting\_array[index]

counting\_array[index]=counting\_array[index]-1

resultant\_array[sorted\_index-1]=index

if(min\_vallue<0):

for x in range(0,len(resultant\_array)):

resultant\_array[x]=resultant\_array[x]+(min\_vallue)

if(sorttype!="Ascending"):

resultant\_array.reverse()

return resultant\_array

**Code in Python**

def counting\_sort(passed\_array,sorttype):

min\_vallue=min(passed\_array)

if(min\_vallue<0):

for x in range(0,len(passed\_array)):

passed\_array[x]=passed\_array[x]-(min\_vallue)

max\_value=max(passed\_array)+1

counting\_array=[0]\*max\_value

resultant\_array=[0]\*len(passed\_array)

for x in range(0,len(passed\_array)):

index=passed\_array[x]

counting\_array[index]=counting\_array[index]+1

for x in range(0,len(counting\_array)-1):

counting\_array[x+1]=counting\_array[x+1]+counting\_array[x]

for x in range(len(passed\_array)-1,-1,-1):

index=passed\_array[x]

sorted\_index=counting\_array[index]

counting\_array[index]=counting\_array[index]-1

resultant\_array[sorted\_index-1]=index

if(min\_vallue<0):

for x in range(0,len(resultant\_array)):

resultant\_array[x]=resultant\_array[x]+(min\_vallue)

if(sorttype!="Ascending"):

resultant\_array.reverse()

return resultant\_array

**Time Complexity**

def counting\_sort(passed\_array,sorttype):

min\_vallue=min(passed\_array)\_\_\_\_\_\_\_\_\_\_1

if(min\_vallue<0):\_\_\_\_\_\_\_\_1

for x in range(0,len(passed\_array)):\_\_\_\_\_\_\_\_\_n+1

passed\_array[x]=passed\_array[x]-(min\_vallue)\_\_\_\_n

max\_value=max(passed\_array)+1

counting\_array=[0]\*max\_value\_\_\_\_\_\_\_\_\_\_\_n+1

resultant\_array=[0]\*len(passed\_array)\_\_\_\_\_\_\_\_\_n

for x in range(0,len(passed\_array)):\_\_\_\_\_\_n+1

index=passed\_array[x]\_\_\_\_\_\_\_\_\_\_\_\_\_\_n

counting\_array[index]=counting\_array[index]+1\_\_\_\_n

for x in range(0,len(counting\_array)-1):\_\_\_\_\_\_\_\_\_\_\_\_n

counting\_array[x+1]=counting\_array[x+1]+counting\_array[x]\_\_\_\_n-1

for x in range(len(passed\_array)-1,-1,-1):\_\_\_\_\_\_\_\_\_n+1

index=passed\_array[x]\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n

sorted\_index=counting\_array[index]\_\_\_\_\_\_\_\_\_n

counting\_array[index]=counting\_array[index]-1\_\_\_\_n

resultant\_array[sorted\_index-1]=index\_\_\_\_\_n

if(min\_vallue<0):\_\_\_\_\_\_\_\_\_\_1

for x in range(0,len(resultant\_array)):\_\_\_\_\_\_\_\_n+1

resultant\_array[x]=resultant\_array[x]+(min\_vallue)\_n

if(sorttype!="Ascending"):\_\_\_\_\_\_\_\_\_\_1

resultant\_array.reverse()

return resultant\_array

Time Complexity Will be O( n )

**Strengths**

1. Time complexity of O(n),Runs faster than other sorting algorithms

**Weakness**

1. Additional memory space is required as it uses additional list or array to store countings and original array if requred.

**Dry Run**

Original Array

|  |  |  |  |
| --- | --- | --- | --- |
| 7 | 2 | 6 | 3 |

Iteration 0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 2 | 2 | 3 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 3 | 0 | 0 |

Iteration 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 2 | 2 | 2 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 3 | 6 | 0 |

Iteration 2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 2 | 2 | 2 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 6 | 0 |

Iteration 3

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 2 | 2 | 2 | 3 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 6 | 7 |

**Shell Sort**

It is like an improved version of insertion sort. The algorithm instead of comparing the adjacent elements compares the elements that are n/2 distance apart in the array where n is the size of the array. When the size of the gap becomes 1, the algorithm works exactly like insertion sort

**Pseudocode**

shellsort(Array,sorttype):

expression1=None

expression2=None

if(sorttype=="Ascending"):

expression1="Array[index+gap]<Array[index]"

expression2="Array[prevous\_indexes - gap] > Array[prevous\_indexes]"

else:

expression1="Array[index+gap]>Array[index]"

expression2="Array[prevous\_indexes - gap] < Array[prevous\_indexes]"

gap=int(len(Array)/2)

while gap>0:

index=0

while (gap+index) < len(Array):

if(eval(expression1)):

Array[index],Array[index+gap]=Array[index+gap],Array[index]

prevous\_indexes=index

while prevous\_indexes - gap >= 0:

if eval(expression2):

Array[prevous\_indexes-gap],Array[prevous\_indexes] = Array[prevous\_indexes],Array[prevous\_indexes-gap]

prevous\_indexes =prevous\_indexes-1

index=index+1

gap=int(gap/2)

return Array

**Code in Python**

def shellsort(Array,sorttype):

expression1=None

expression2=None

if(sorttype=="Ascending"):

expression1="Array[index+gap]<Array[index]"

expression2="Array[prevous\_indexes - gap] > Array[prevous\_indexes]"

else:

expression1="Array[index+gap]>Array[index]"

expression2="Array[prevous\_indexes - gap] < Array[prevous\_indexes]"

gap=int(len(Array)/2)

while gap>0:

index=0

while (gap+index) < len(Array):

if(eval(expression1)):

Array[index],Array[index+gap]=Array[index+gap],Array[index]

prevous\_indexes=index

while prevous\_indexes - gap >= 0:

if eval(expression2):

Array[prevous\_indexes-gap],Array[prevous\_indexes] = Array[prevous\_indexes],Array[prevous\_indexes-gap]

prevous\_indexes =prevous\_indexes-1

index=index+1

gap=int(gap/2)

return Array

**Time Complexity**

def shellsort(Array,sorttype):

expression1=None\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

expression2=None\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

if(sorttype=="Ascending"):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

expression1="Array[index+gap]<Array[index]"

expression2="Array[prevous\_indexes - gap] > Array[prevous\_indexes]"

else:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

expression1="Array[index+gap]>Array[index]"

expression2="Array[prevous\_indexes - gap] < Array[prevous\_indexes]"

gap=int(len(Array)/2)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

while gap>0:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n/2+1

index=0\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n/2

while (gap+index) < len(Array):\_\_\_\_\_\_\_\_\_n/2(n/2+1)

if(eval(expression1)):\_\_\_\_\_\_\_\_\_\_\_\_\_n/2(n/2)

Array[index],Array[index+gap]=Array[index+gap],Array[index]\_\_\_\_\_\_\_\_\_\_\_\_\_n/2(n/2)

prevous\_indexes=index\_\_\_\_\_\_\_\_\_\_\_\_\_n/2(n/2)

while prevous\_indexes - gap >= 0:\_\_\_\_\_\_\_n/2(n/2)

if eval(expression2):\_\_\_\_\_\_\_\_\_\_\_\_\_n/2(n/2)

Array[prevous\_indexes-gap],Array[prevous\_indexes] = Array[prevous\_indexes],Array[prevous\_indexes-gap]\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n/2(n/2)

prevous\_indexes =prevous\_indexes-1\_\_\_n/2(n/2)

index=index+1\_\_\_\_n/2(n/2)

gap=int(gap/2)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n/2

return Array

Time Complexity Will be O(n^2) in worst case

**Strengths**

1. Is in Inplace sorting Algorithm
2. It has the time time complexity of O(nlogn) in best case faster than some other sorting algorithms

**Weakness**

1. It is not a stable sort algorithm
2. It is still slow than some others algorithm its worst case is O( n^2 )

**Dry Run**

Original Array

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 6 | 3 |

Iteration 0

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 6 | 3 |

Iteration 1

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 3 | 6 |

Iteration 2

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 3 | 6 |

**Comb Sort**

It is an improved version of bubble sort where instead of comparing adjacent values, the values compared are n/1.3 distance apart in array til n becomes 1 and the algorithm behaves exactly like bubble sort

**Pseudocode**

def Combsort(Array,sorttype):

expression=None

if(sorttype!="Ascending"):

expression="Array[index] < Array[gap+index]"

else:

expression="Array[index] > Array[gap+index]"

gap=int(len(Array))

while gap>0:

index=0

while (gap+index)<len(Array):

if eval(expression) :

Array[index],Array[index+gap]=Array[index+gap],Array[index]

index=index+1

gap=int(gap/1.3)

**Code in Python**

def Combsort(Array,sorttype):

expression=None

if(sorttype!="Ascending"):

expression="Array[index] < Array[gap+index]"

else:

expression="Array[index] > Array[gap+index]"

gap=int(len(Array))

while gap>0:

index=0

while (gap+index)<len(Array):

if eval(expression) :

Array[index],Array[index+gap]=Array[index+gap],Array[index]

index=index+1

gap=int(gap/1.3)

return Array

**Time Complexity**

Combsort(Array,sorttype):

expression=None\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

if(sorttype!="Ascending"):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

expression="Array[index] < Array[gap+index]"

else: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

expression="Array[index] > Array[gap+index]"

gap=len(Array)#Get the length of the Array\_\_\_\_\_\_\_\_\_\_1

while gap>0:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ n/1.3+1

index=0\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ n/1.3

while (gap+index)<len(Array):\_\_\_\_\_\_\_n/1.3( n/1.3+1)

if eval(expression) :\_\_\_\_\_\_\_\_\_\_\_\_\_ n/1.3( n/1.3+1)

Array[index],Array[index+gap]=Array[index+gap],Array[index]\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ n/1.3( n/1.3+1)

index=index+1\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ n/1.3( n/1.3+1)

gap=int(gap/1.3)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ n/1.3

return Array

In Worst Case Scenario Time Complexity Will be O(n^2)

**Strengths**

1. Is in Inplace sorting Algorithm
2. It has the best case time time complexity of O(nlogn)

**Weakness**

1. It is not a stable sort algorithm
2. It is still slow than some others algorithm as it has time complexity of O(n^2)

**Dry Run**

Original Array

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 6 | 3 |

Iteration 0

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 6 | 3 |

Iteration 1

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 6 | 3 |

Iteration 2

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 6 | 3 |

Iteration 3

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 6 | 3 |

Iteration 4

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 6 | 3 |

Iteration 5

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 3 | 6 |

Iteration 6

|  |  |  |  |
| --- | --- | --- | --- |
| -1 | 2 | 3 | 6 |

**Cycle Sort**

All of the elements starting from the first index of an array are cycled through the indices till their correct position is found where they are then swapped. The algorithm finds the position by determining whether the next item is greater than or not. When the next element is greater, the element is swapped at a position before it.

**Pseudocode**

cyclesort(Array,sorttype):

expression=None

if(sorttype=="Ascending"):

expression="Array[x]<value"

else:

expression="Array[x]>value"

for index=0 to len(Array):

value=Array[index]

current\_index=index

for x=index+1 to len(Array):

if(eval(expression)):

current\_index=current\_index+1

if(current\_index!=index):

while value==Array[current\_index]:

current\_index=current\_index+1

Array[current\_index],value=value,Array[current\_index]

while current\_index!=index:

current\_index=index

for x in range(index+1,len(Array)):

if(eval(expression)):

current\_index= current\_index+1

while value==Array[current\_index]:

current\_index=current\_index+1

Array[current\_index],value=value,Array[current\_index]

return Array

**Code in Python**

def cyclesort(Array,sorttype):

expression=None

if(sorttype=="Ascending"):

expression="Array[x]<value"

else:

expression="Array[x]>value"

for index in range(0,len(Array)):

value=Array[index]

current\_index=index

for x in range(index+1,len(Array)):

if(eval(expression)):

current\_index=current\_index+1

if(current\_index!=index):

while value==Array[current\_index]:

current\_index=current\_index+1

Array[current\_index],value=value,Array[current\_index]

while current\_index!=index:

current\_index=index

for x in range(index+1,len(Array)):

if(eval(expression)):

current\_index= current\_index+1

while value==Array[current\_index]:

current\_index=current\_index+1

Array[current\_index],value=value,Array[current\_index]

return Array

**Time Complexity**

def cyclesort(Array,sorttype):

expression=None\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

if(sorttype=="Ascending"):\_\_\_\_\_1

expression="Array[x]<value"

else:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

expression="Array[x]>value"

for index in range(0,len(Array)): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n+1

value=Array[index]\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n

current\_index=index\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n

for x in range(index+1,len(Array)):\_\_\_\_\_\_\_\_\_\_\_n(n+1)

if(eval(expression)):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_n(n)

current\_index=current\_index+1\_\_\_\_\_n(n)

if(current\_index!=index):\_\_\_\_\_\_\_\_\_\_\_\_n(n)

while value==Array[current\_index]:

current\_index=current\_index+1

Array[current\_index],value=value,Array[current\_index]

while current\_index!=index:

current\_index=index

for x in range(index+1,len(Array)):

if(eval(expression)):

current\_index= current\_index+1

while value==Array[current\_index]:

current\_index=current\_index+1

Array[current\_index],value=value,Array[current\_index]

return Array

Time Complexity Will Be O(n^2) in worst case

Time Complexity Will Be O(n^2) in Best case

Time Complexity Will Be O(n^2) in Average case

**Strengths**

1. It is an in-place sorting algorithm thus does not require additional memory.

**Weakness**

1. It is an unstable sorting algorithm
2. It has the time complexity of O(n^2) Thus slower than other algorithms.

**Dry Run**

Original Array

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 6 | 3 | 8 | 5 | 1 | 4 |

Iteration 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 8 | 5 | 6 | 4 |

Iteration 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 8 | 5 | 6 | 4 |

Iteration 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 8 | 5 | 6 | 4 |

Iteration 3

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 8 |

Iteration 4

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 8 |

Iteration 5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 8 |

Iteration 6

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 8 |

**Searching Algorithms**

The following algorithms are provided with the column in which they need to search for the key which is also passed as parameter. The code then returns the indices of all the occurrences in the given column.

**Linear Search**

def linear\_search(AllObjects,key,column):

print(len(key))

expression = "(getattr(AllObjects[i],column[j]))"

occurences = []

for j in range(len(key)):

for i in range(len(AllObjects)):

string=str(eval(expression))

if (key[j]) in string:

occurences.append(i)

return occurences

**Binary Search**

def binary\_search\_counter(A,low,high,key):

occur\_binary = []

index = binary\_search(A,low,high,key)

if index == -1:

return None

occur\_binary.append(index)

left = index-1

while left>=0 and A[left]==key:

occur\_binary.append(left)

left-=1

right = index+1

while right<len(A) and A[right]==key:

occur\_binary.append(right)

right+=1

occur\_binary.sort()

return occur\_binary

def binary\_search(A,low,high,key):

if low!=high or key == A[low]:

mid = (low+high)//2

if A[mid] == key:

return mid

elif key<A[mid]:

return binary\_search(A,low,mid,key)

else:

return binary\_search(A,mid+1,high,key)

return -1

**Jump Search**

def linearJump\_search(A,end,key):

occurences = []

for i in range(end+1):

if A[i] == key:

occurences.append(i)

return occurences

def jump\_search(A,key):

jumpsize = int(math.sqrt(len(A)))

#print("jump:",jumpsize)

end = 0

found = False

i = 0

while i<=len(A)-1 and found == False:

if A[i]>key and i>=jumpsize:

end = i

found = True

i+=jumpsize

print(i)

if found == False:

if A[len(A)-1]==key:

end = len(A)-1

found = True

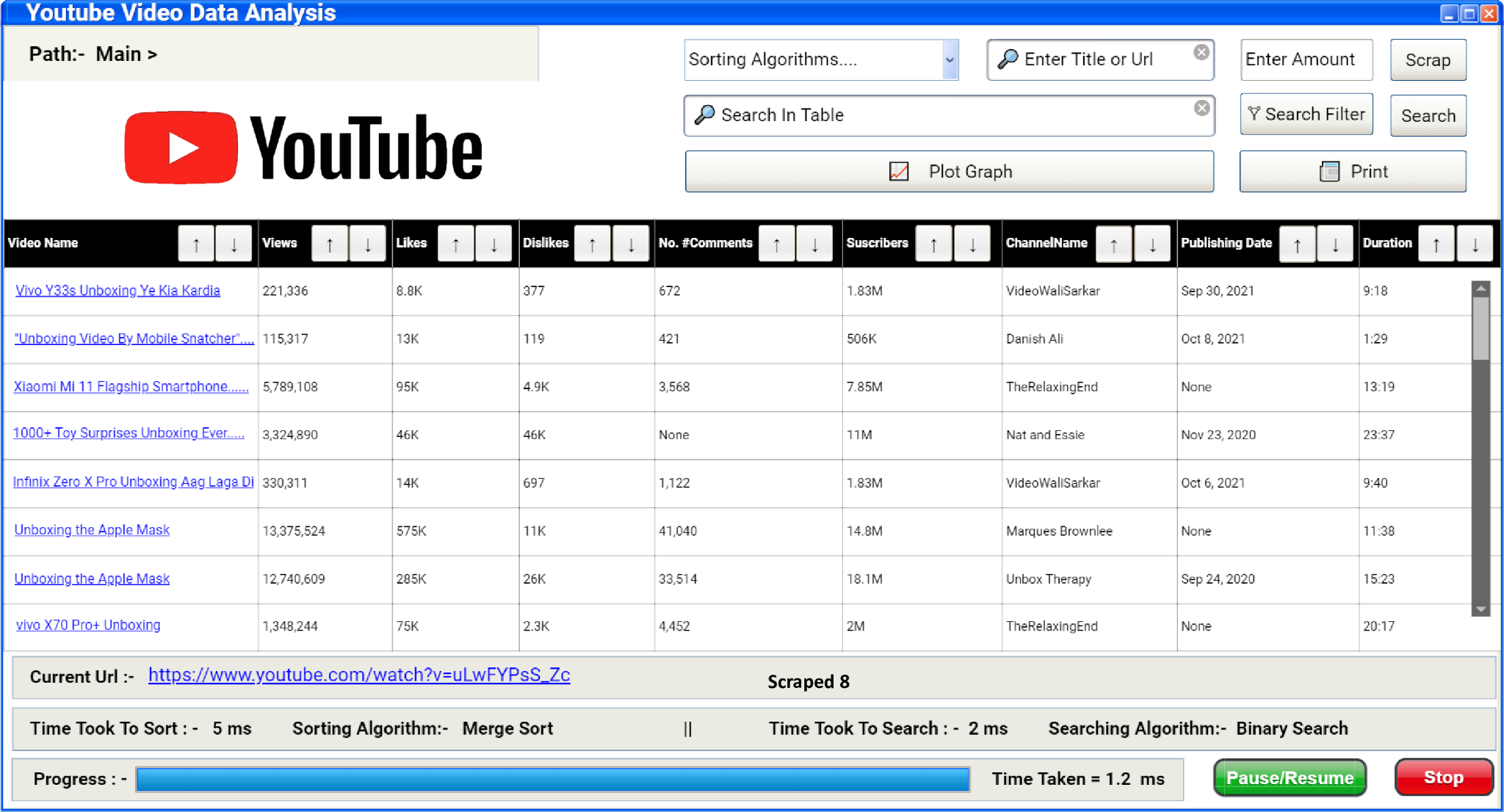
return linearJump\_search(A,end,key)

**UI**

**Initial GUI design**

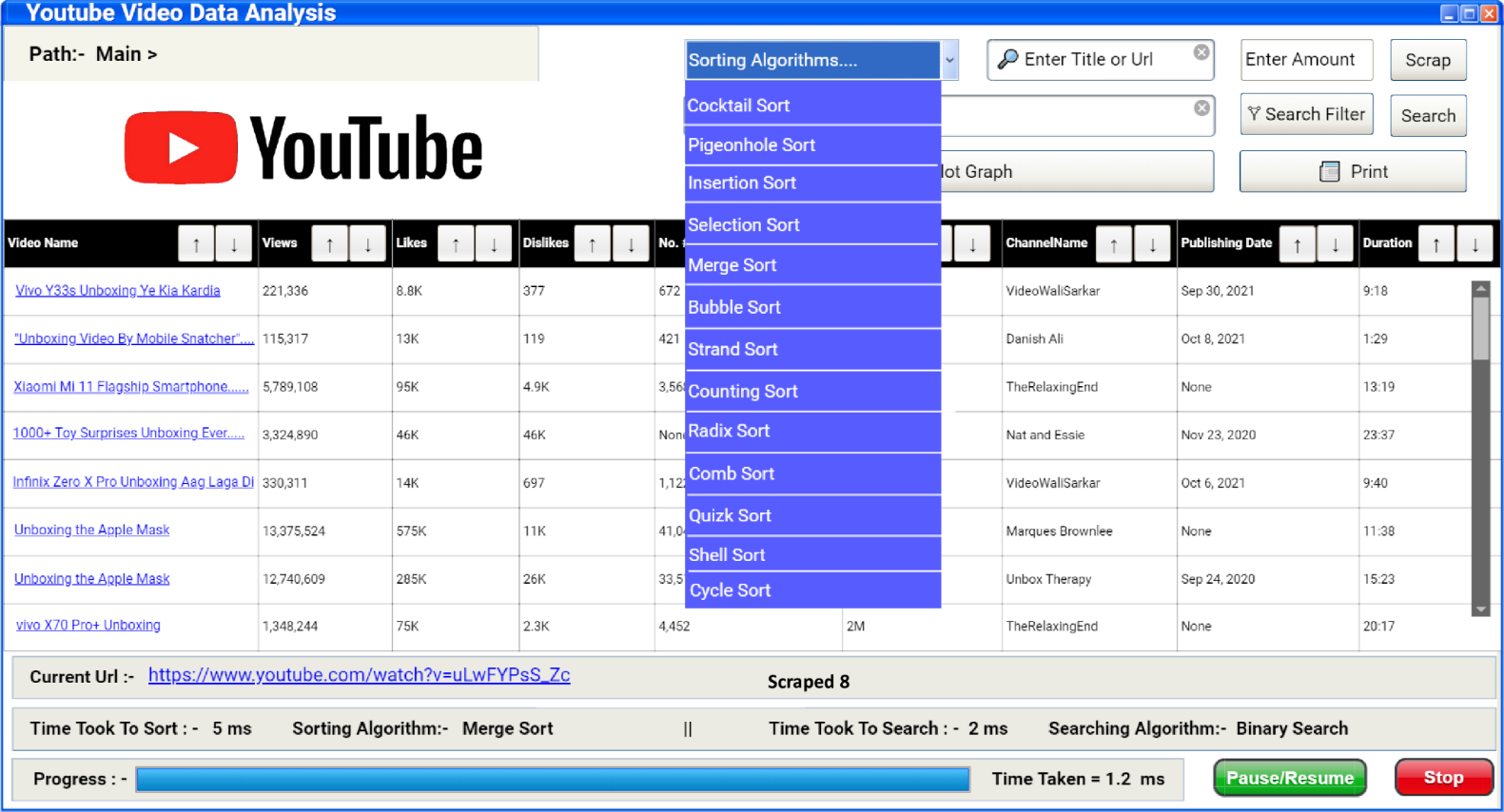
The following design was implemented using the pencil tool.

Main Application Screen

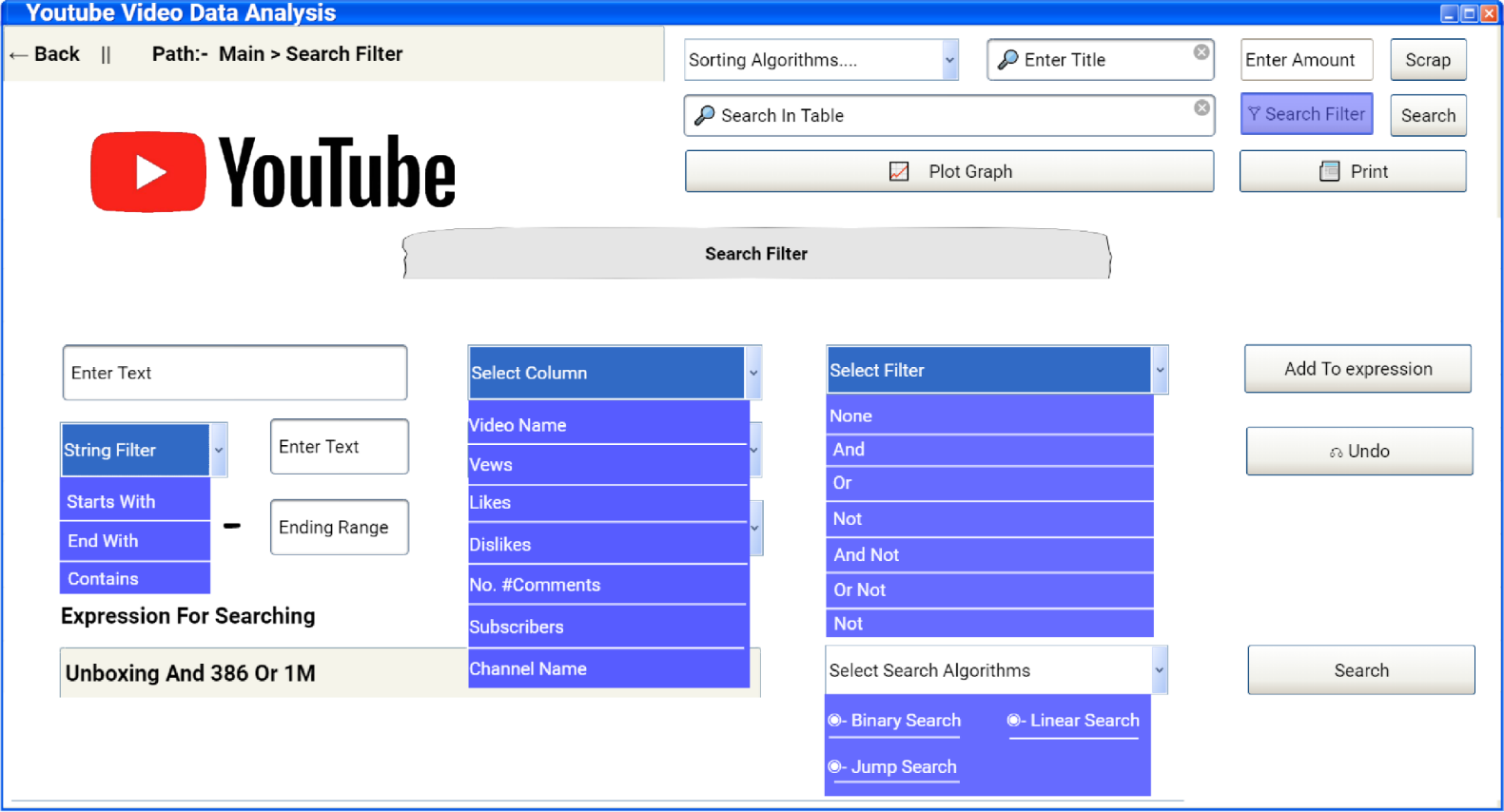


***Figure 1***

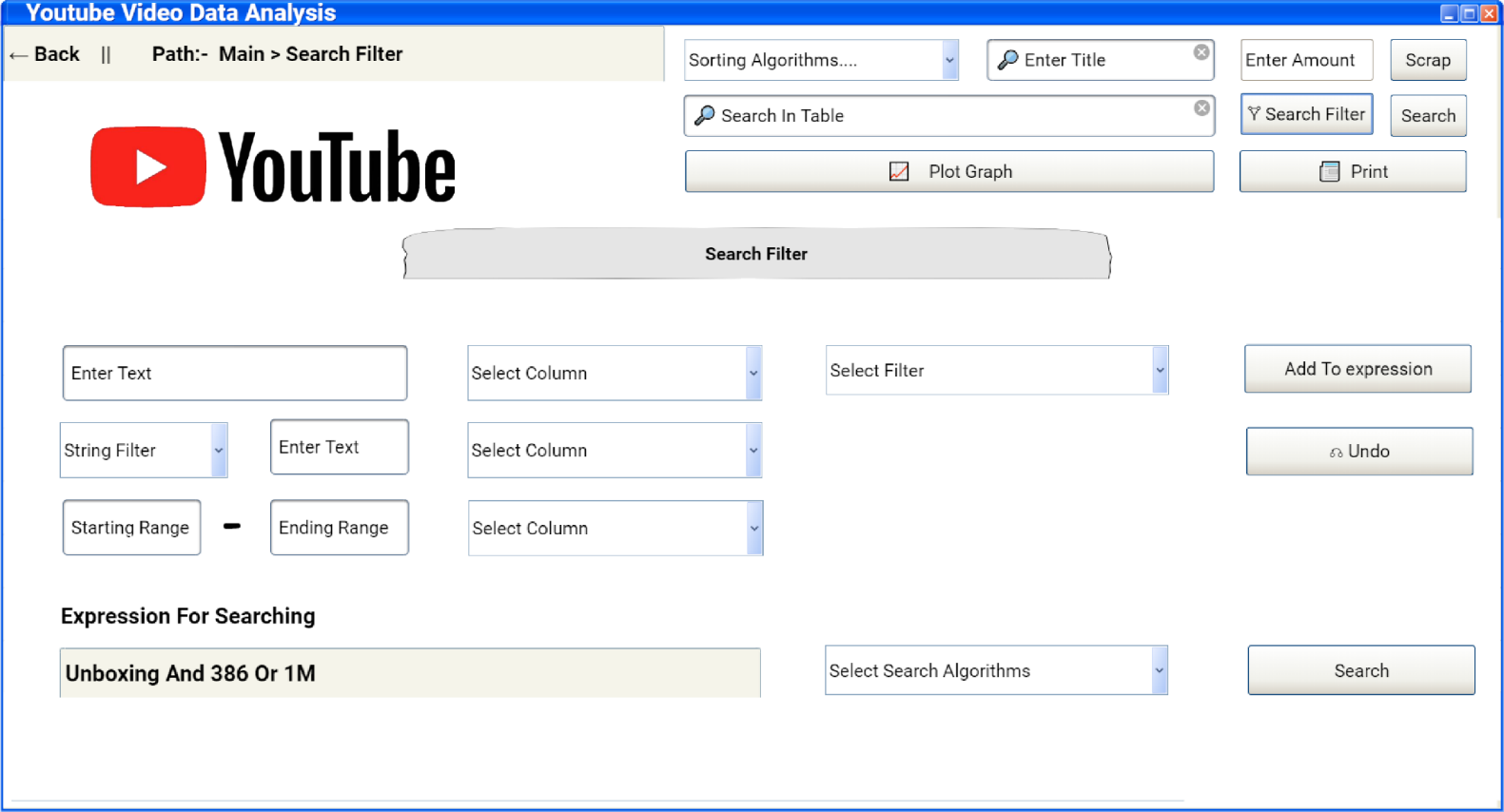
|  |  |  |
| --- | --- | --- |
| **UI Component Name** | **Type of component** | **Purpose** |
| Main Table | Table | Used to display all the data scraped by the program. |
| Information Bar | Status Bar | Informs us about which URL is currently under process, How much time it took to sort by the selected algorithm and how much it took to search by selected search algorithm. |
| VideoNames | Hyperlink | It is the component of Main Table it displays the video name but Hyperlink is encoded in it. |
| ↑ Sort Desendingly | Button | So that user can sort the column in descending order. |
| ↓ Sort Ascendingly | Button | So that user can sort the column in  ascending order. |
| Sorting Algorithms | DropDown | So that user can select which algorithm they want to use to sort columns |
| Enter Title or Url | Search Bar | When user enters the title, amount and clicks on “scrap” all the videos with that title is scraped by the program or if the user gives a URL the video is scraped by the program |
| Enter Amount | Text Box | To take input of the amount of videos that the user wants scrapped by the user provided URL. |
| Scrap | Button | A button that will start scrapping from the user provided URL |
| Search in Table | Text Box | This allows the user to enter any string or integer in the whole table which then will be highlighted without any filters |
| Plot Graph | Button | It will open a window that will allow the user to plot graphs of selected quantity |
| Search | Button | It will search for the query input by user in “SEARCH IN TABLE” textbox |
| Search Filter | Button | A button that will open an interface  allowing the user to manipulate the searching filters according to need. |
| Print | Button | When it is pressed a csv or excel file is generated and a print request is sent to the printer to print that csv file or excel file |
| Progress Bar | Progress Bar | Shows percentage of how many items  have been scrapped from the total |
| Pause/Resume | Button | This allows the user to pause the scrapping that is under process or resume the scrapping from the point onward it was paused at. |
| Stop | Button | This Terminates the process of scrapping entirely and so scrapping will not resume but will restart if the user wishes to resume the scrapping |
| Enter Text | Text Box | Used to enter text by the user so that he can apply AND OR NOT filter on the column by this word |
| Select Column | Drop Down | Used to select the column that is to be  searched by the user |
| Select Filter | Drop Down | So that the user can select the filter he wants to apply which include AND, OR, NOT, AND NOT, OR NOT. |
| Add expression | Button | This button is used to add expression for searching for example if user enters book then select AND filter then click on it will add expression book And in expression block below that will be used for search |
| String Filter | Drop Down | Used so that the user can select which string filter he wants to use in his search i.e Starts with, Ends with and Contains. |
| Starting Range  -  Ending Range | Text Box | User can specify the starting and ending range in his search for integers and date data types |
| Search | Button | When pressed the query which is selected after applying all the filters is searched in the table and the table is highlighted where it appears or word of it appears |
| Undo | Button | Removes the last expression added in the expression table |
| Select Search Algorithm | Drop Down | So that user can select which search  algorithm the user wants to use for searching purpose |
| Select Graph Type | Drop Down | User can select which type of graph they want to plot |
| Graph | Panel | Is used to display the graph plotted by  the program |
| Back | Button | Used to return to the main page |



***Figure 2***

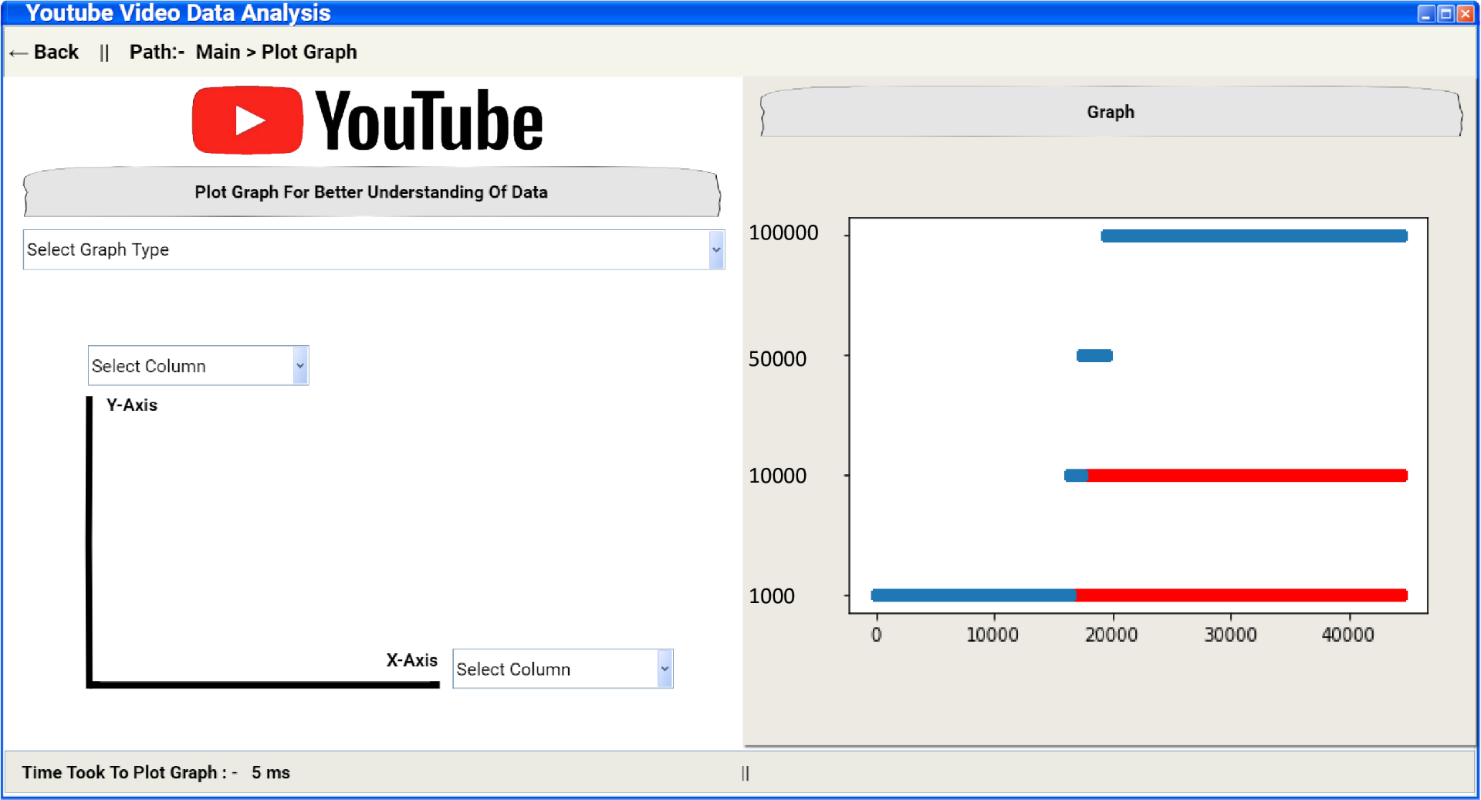
Search Screen

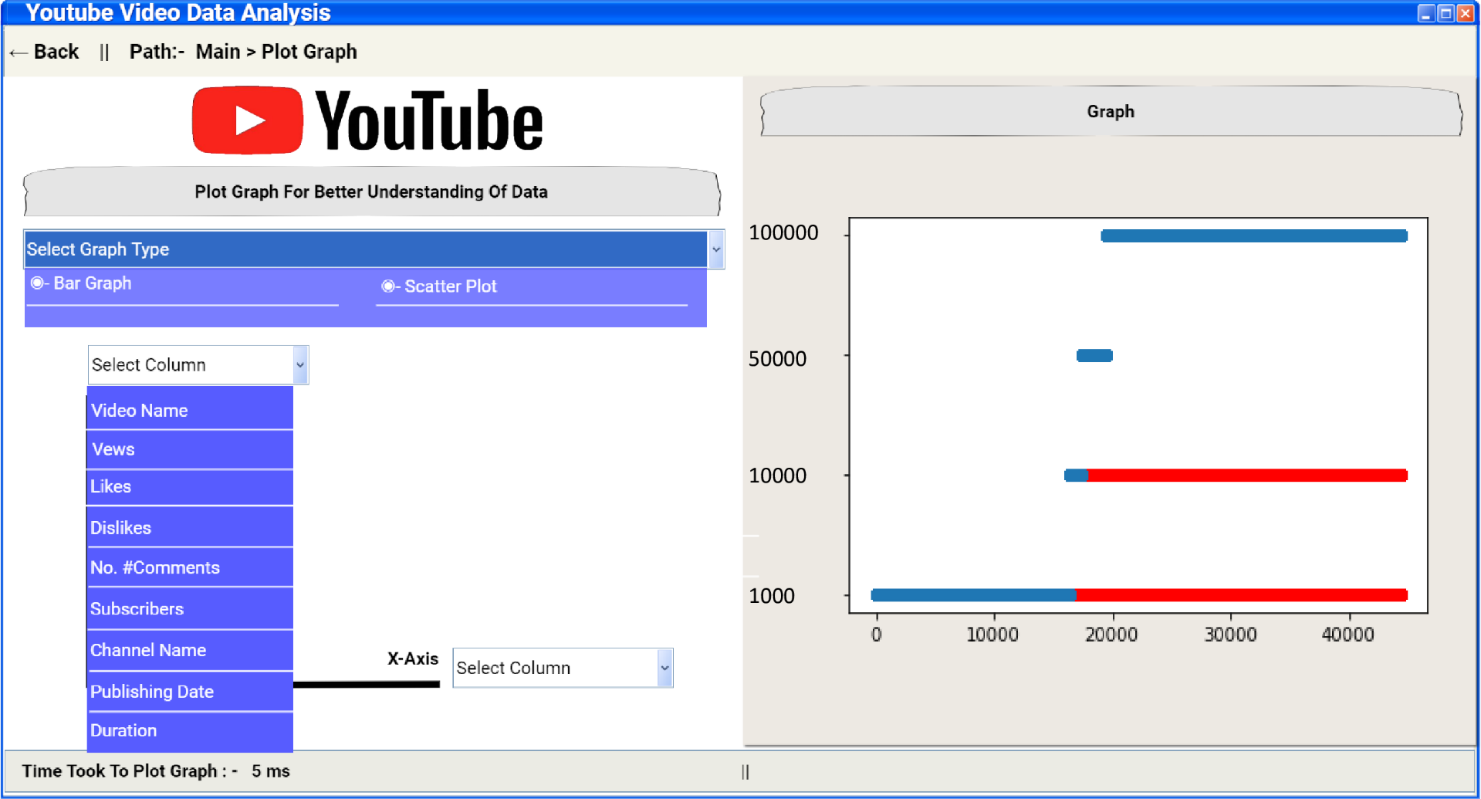
***Figure 3***



***Figure 4***

Graph Screen



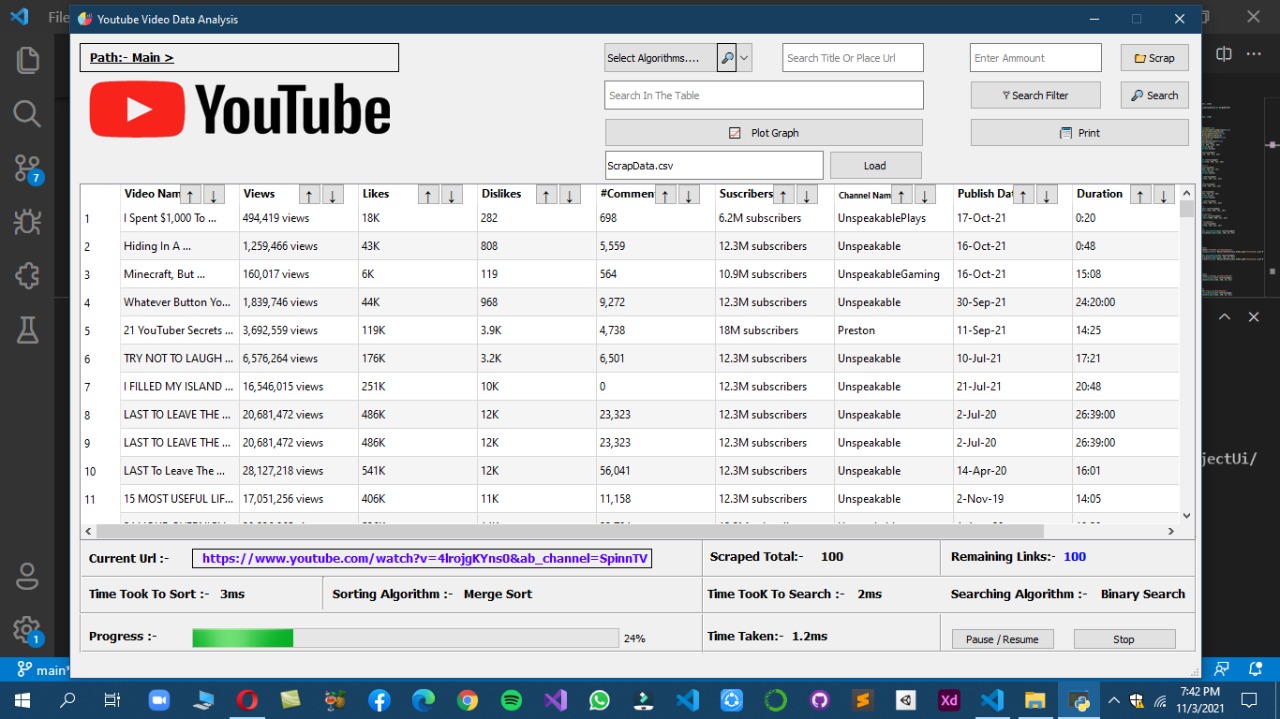
***Figure 5***

***Figure 6***

**Implemented GUI**

The GUI was implemented using the QT designer and PyQt5 and its respective tools library. There were a few changes in the actual implementation because in the initial design a very crucial button was missed which had to load all the previous data into the application.

Main Screen

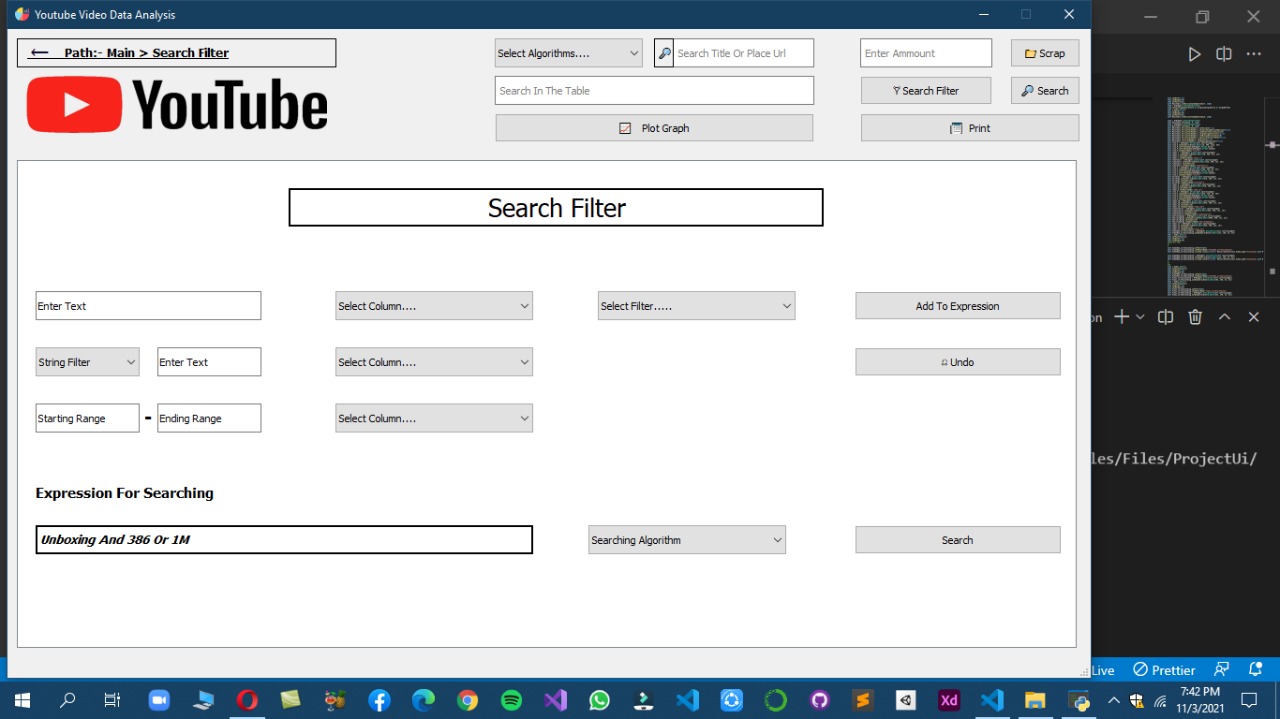


***Figure 7***

As it can be seen in the ***Figure 7*** a button has been inserted and a text box below the plot graph button. This now allows the user to enter a csv file name corresponding to the project columns which then will be loaded in the application table

The information bar at the bottom of the screen has also been slightly modified by placing the labels and information in a more organized manner allowing easier comprehension of information.

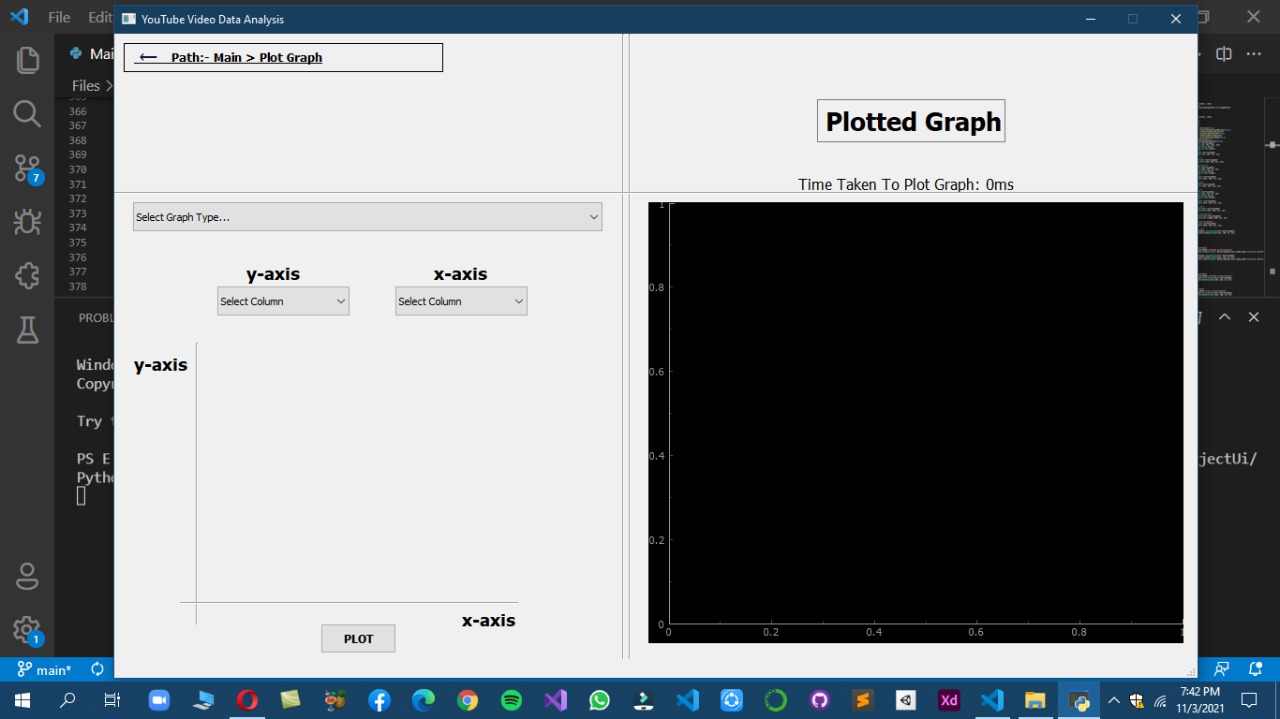
Search Screen



***Figure 8***

The search screen UI had no changes and has been implemented as it is.

Graph Screen



***Figure 9***

The graph plotting screen has had a minor change as to the graph being used. In the GUI the graph was a pandas’ graph feature but here we have implemented the PyQt5 Graph which is more interactive as it allows zooming in and out. The time label which shows the time taken to plot the graph has been moved above the graph.

**Integration**

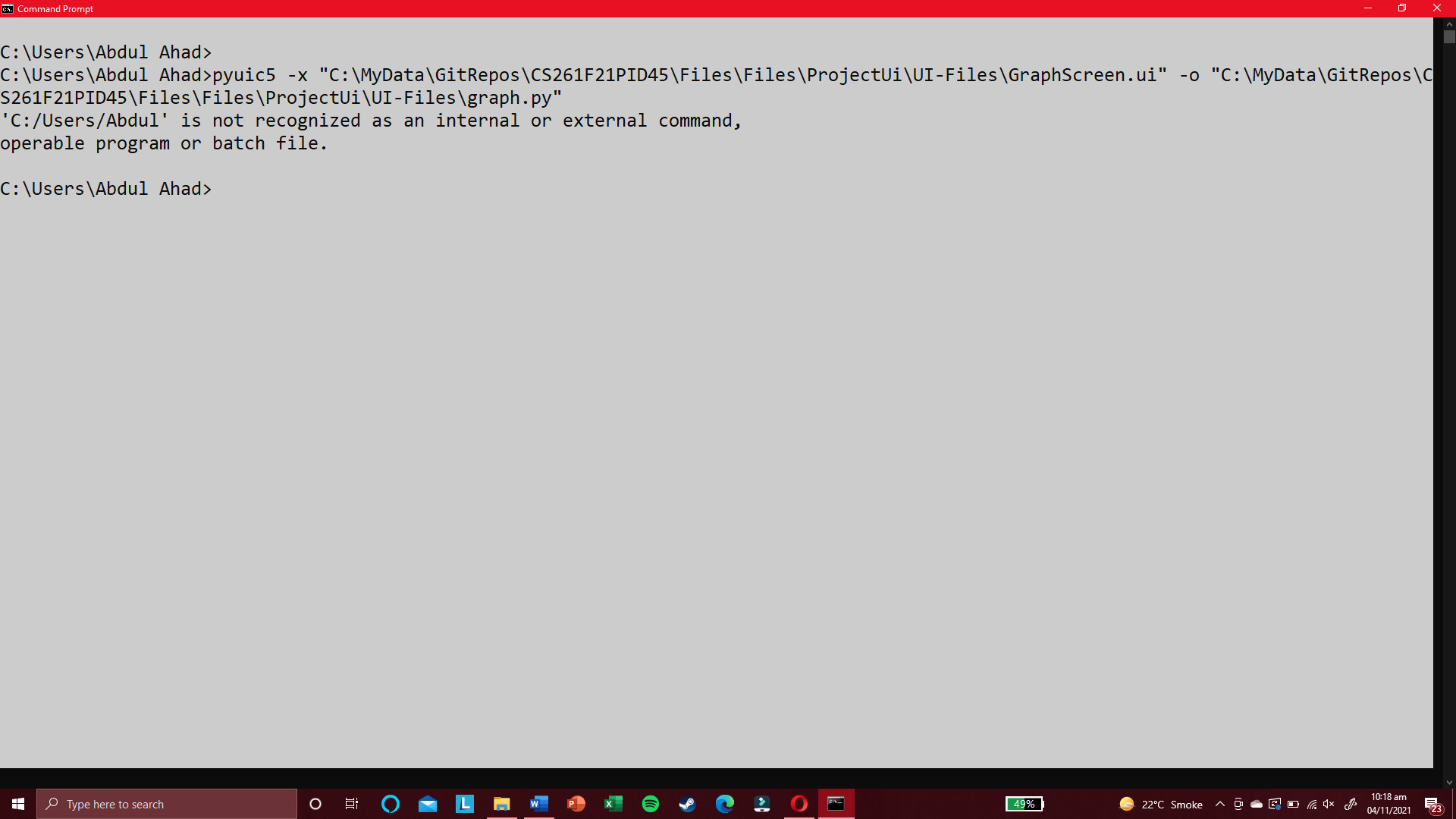
**Problems and their solution**

Due to a lack of preplanning and miscommunication, we had no clue as to where to begin the integration. Moreover, we had a lot of loopholes in our code since day one. Due to a lack of knowledge on web scraping, the data that we scraped had a lot of discrepancies and anomalies which became a huge problem during integration.

The first major problem that we faced in our project was finding the tags while scraping. The tags on the YouTube page were unusually complex but then again being just a novice, we did not have much of an experience to scrape the data quickly. The HTML tags and classes were the same for a lot of the data present. Using the “FindNext” Command we scraped the data for the selected attributes we required from the video.

The next immediate problem was that the number of comments wouldn’t load because they only load when the user scrolls the page down. Naturally, it took a lot of time in debugging the problem. We then surfed the internet to conclude that we had to scroll the page using the web driver. The problem still wasn’t resolved. The underlying issue was not only that the comments wouldn’t load until the page was scrolled but also due to low internet bandwidth, the page wouldn’t load as quickly as the code ran and so we had to add a sleep delay to the system to allow the page to first load fully and then scrape data otherwise it’d just scrape the data that’d load at the first instance saving several null values in the file. We had to submit the data for 1 million records, we didn’t have much of a choice but to save and submit whatever we had, even if the data was anomalous.

After the scraping process, we moved to the UI implementation where one of the members worked while the other had issues with his system. The first and foremost problem was the multiple instances of python and its’ libraries being present on the system due to which the system failed to understand which python version it needed to use. The extra instances were then deleted to resolve this error.

The Qt designer had to be understood before being used. Although we had previously worked on the Windows forms using Visual Studio and C#, Qt designer was quite different. The work was distributed but due to path not being recognized on one of the members’ systems led to a delay in UI Implementation. The problem was that the python file for the designed UI wouldn’t generate because the Username was set to “Abdul Ahad” which had a space in between. Even though the full path was given, the command prompt failed to find the directory. 

We searched the internet but we couldn’t take a risk of changing the path so that it would work because, in case of a failure, all the work in the future could be adversely affected and halt our already impeded progress. After several hours of consideration, a solution was then suggested by a fellow class member which was to create a new user account on the system and create a python file from there. Once the file was created the three UI screens could only be connected to scroll between in, due to shortage of time.

The UI was implemented partially as all the buttons were not yet enabled or connected; we began working on the sorting algorithms. Due to a lack of critical analysis on our end, we failed to realize that the sorting algorithm had to be written so that later when used they can also sort for strings. We wrote the sorting algorithms which only sported a given array and returned that sorted array. This created a problem later. By the time we started integrating all the components, we realized that we had to use Object-Oriented programming and all the sorting algorithms we had written were rendered useless as they only sorted a single column and not the whole records as they should. Furthermore, we had little to no knowledge of the syntax of the PyQt5 and the widgets. We then had to watch tutorials and learn and implement within the few days we had. We re-designed all the structures of our program to convert to the Object-Oriented paradigm.

The major task in all of this endeavor was extracting the data from the data that was scrapped initially because there were a lot of different formats for how the strings were saved in the CSV file. There was also duplication in data but we removed it by using a feature of Microsoft excel to delete the duplicate rows. 0.78% records were repeated in the whole data.

All these problems created a lot of pressure and workload. The time to implement became shorter by the minute and so we failed to implement several features.

**Overview of Working**

The application loads up and an interface for main screen is shown. When a file is loaded into the table using the .csv extension in name, all of the records are first turned into objects and then stored in lists. These lists then become the basis for sorting and searching. The sorting functions use these lists and the column that is being sorted to sort the whole table. As for the graphing function, it first reads the default file and plots the graph depending on the type of graph selected by the user and columns selected for x and y axis.

**Features**

* User can load the data from his/her own file but it must strictly be .csv and related to this application
* The user can print the file
* The user can plot the graph of choice using the selective columns
* The user can scrape data from a YouTube link
* The user can also scrape data from a keyword in the same text box. The user has to give a certain number of links that will then be scraped from the keyword.
* The total number of records that contain the searched keyword will be shown in the information bar below and the respective records are highlighted as red in the table.
* User can freely scroll between the three windows namely main screen, graph screen, and search screen.
* **DOUBLE** clicking on the video name in the table will open the respective video as well.

**Missing features**

Due to the problems highlighted above we were unable to implement a few features. Although the options are shown in the GUI, we haven’t yet programmed those features.

* Multi-Level Sorting
* Multi-Level Searching

The user can search multiple columns but the program is designed only to search using OR keyword by default.

* Search Filters for instance starts with, contains, ends with and range
* Undo button that’d remove the last expression added for searching.
* Pause/Resume, Stop button and Progress Bar. Due to lack of understanding on multi-threading we were unable to implement these features.
* Radix Sort doesn’t work for strings
* Bar graph has not been implemented
* The duration cannot be plotted on graphs
* The current link that is being scraped cannot be changed as a side effect of multithreading nor being implemented at all.

**Collaboration**

We divided the work between ourselves. The deliverable was divided in half initially but then considering the pace of the other member in case of a problem, the work load was then shouldered if the other member completed his part earlier. On weekends, we had multiple daily meetings on Google Meet to collaborate on how and what to work on. Moreover, we skipped lectures of other courses to complete the leftover work. We sat in library to discuss the issues in our program and work we had divided. We stayed in hostels despite being day scholars. We worked overtime and overnight. We used the GitHub repository to commit our codes so that our codes would then merge and not cause any kind of conflict. We communicated via messenger before committing the files and made backups to prevent data loss.