|  |
| --- |
| e-Books Retrieval System |
| Data Structures and Algorithms (Mid Term Project) |

|  |
| --- |
| Submitted to: Sir Samyan Qayyum Wahla  By: CS261F21PID36  Group Member 01: Moiz-ul-Haq 2020-CS-19  Group Member 02: Mubashar Ahmad 2020-CS-34 |

Table of Contents

[1 Motivation: 7](#_Toc87026502)

[2 Target Audience: 7](#_Toc87026503)

[3 Business Need: 7](#_Toc87026504)

[4 Data (Scrapping): 8](#_Toc87026505)

[5 Sorting Algorithms: 9](#_Toc87026506)

[5.1 Selection Sort: 9](#_Toc87026507)

[5.1.1 Description: 9](#_Toc87026508)

[5.1.2 Pseudo Code: 9](#_Toc87026509)

[5.1.3 Code in Python: 9](#_Toc87026510)

[5.1.4 Time Complexity Analysis: 9](#_Toc87026511)

[5.1.5 Strengths: 9](#_Toc87026512)

[5.1.6 Weakness: 9](#_Toc87026513)

[5.1.7 Dry run on small input: 10](#_Toc87026514)

[5.2 Insertion sort: 10](#_Toc87026515)

[5.2.1 Description: 10](#_Toc87026516)

[5.2.2 Pseudo Code: 10](#_Toc87026517)

[5.2.3 Code in Python: 10](#_Toc87026518)

[5.2.4 Time Complexity Analysis: 11](#_Toc87026519)

[5.2.5 Strengths: 11](#_Toc87026520)

[5.2.6 Weakness: 11](#_Toc87026521)

[5.2.7 Dry run on small input: 12](#_Toc87026522)

[5.3 Bubble Sort: 12](#_Toc87026523)

[5.3.1 Pseudo Code: 12](#_Toc87026524)

[5.3.2 Code in Python: 12](#_Toc87026525)

[5.3.3 Time Complexity Analysis: 13](#_Toc87026526)

[5.3.4 Strengths: 13](#_Toc87026527)

[5.3.5 Weakness: 13](#_Toc87026528)

[5.3.6 Dry run on small input: 14](#_Toc87026529)

[5.4 Merge Sort: 14](#_Toc87026530)

[5.4.1 Description: 14](#_Toc87026531)

[5.4.2 Pseudo Code: 14](#_Toc87026532)

[5.4.3 Code in Python: 15](#_Toc87026533)

[5.4.4 Time Complexity Analysis: 17](#_Toc87026534)

[5.4.5 Strengths: 17](#_Toc87026535)

[5.4.6 Weakness: 17](#_Toc87026536)

[5.4.7 Dry run on small input: 18](#_Toc87026537)

[5.5 Cycle Sort: 18](#_Toc87026538)

[5.5.1 Description: 18](#_Toc87026539)

[5.5.2 Pseudo Code: 19](#_Toc87026540)

[5.5.3 Code in Python: 19](#_Toc87026541)

[5.5.4 Proof of correctness: 20](#_Toc87026542)

[5.5.5 Strengths: 20](#_Toc87026543)

[5.5.6 Weakness: 20](#_Toc87026544)

[5.6 Bitonic Sort: 21](#_Toc87026545)

[5.6.1 Description: 21](#_Toc87026546)

[5.6.2 Pseudo Code: 21](#_Toc87026547)

[5.6.3 Code in Python: 21](#_Toc87026548)

[5.6.4 Time Complexity Analysis: 22](#_Toc87026549)

[5.6.5 Three strengths: 23](#_Toc87026550)

[5.6.6 Three weakness: 23](#_Toc87026551)

[5.6.7 Dry run on small input: 23](#_Toc87026552)

[5.7 Heap Sort: 23](#_Toc87026553)

[5.7.1 Pseudo Code: 23](#_Toc87026554)

[5.7.2 Code in Python: 24](#_Toc87026555)

[5.7.3 Time Complexity Analysis: 25](#_Toc87026556)

[5.7.4 Three strengths: 25](#_Toc87026557)

[5.7.5 Three weakness: 25](#_Toc87026558)

[5.7.6 Dry run on small input: 25](#_Toc87026559)

[5.8 Bucket Sort: 25](#_Toc87026560)

[5.8.1 Description: 25](#_Toc87026561)

[5.8.2 Pseudo Code: 26](#_Toc87026562)

[5.8.3 Code in Python: 26](#_Toc87026563)

[5.8.4 Time Complexity Analysis: 27](#_Toc87026564)

[5.8.5 Three Strengths: 27](#_Toc87026565)

[5.8.6 Three Weaknesses: 28](#_Toc87026566)

[5.8.7 Dry Run: 28](#_Toc87026567)

[5.9 Radix Sort: 29](#_Toc87026568)

[5.9.1 Description: 29](#_Toc87026569)

[5.9.2 Pseudo Code: 29](#_Toc87026570)

[5.9.3 Code in Python: 29](#_Toc87026571)

[5.9.4 Time Complexity Analysis: 30](#_Toc87026572)

[5.9.5 Three Strengths: 30](#_Toc87026573)

[5.9.6 Three Weaknesses: 30](#_Toc87026574)

[5.9.7 Dry Run: 31](#_Toc87026575)

[5.10 Quick Sort: 31](#_Toc87026576)

[5.10.1 Description: 31](#_Toc87026577)

[5.10.2 Pseudo Code: 31](#_Toc87026578)

[5.10.3 Code in Python: 32](#_Toc87026579)

[5.10.4 Time Complexity Analysis: 32](#_Toc87026580)

[5.10.5 Three Strengths: 33](#_Toc87026581)

[5.10.6 Three Weaknesses: 33](#_Toc87026582)

[5.10.7 Dry Run: 33](#_Toc87026583)

[5.11 Count Sort: 33](#_Toc87026584)

[5.11.1 Description: 33](#_Toc87026585)

[5.11.2 Pseudo Code: 34](#_Toc87026586)

[5.11.3 Code in Python: 34](#_Toc87026587)

[5.11.4 Time Complexity Analysis: 35](#_Toc87026588)

[5.11.5 Three Strengths: 35](#_Toc87026589)

[5.11.6 Three Weaknesses: 35](#_Toc87026590)

[5.11.7 Dry Run: 36](#_Toc87026591)

[5.12 Pancake Sort: 36](#_Toc87026592)

[5.12.1 Description: 36](#_Toc87026593)

[5.12.2 Pseudo Code: 36](#_Toc87026594)

[5.12.3 Code in Python: 36](#_Toc87026595)

[5.12.4 Time Complexity Analysis: 38](#_Toc87026596)

[5.12.5 Three Strengths: 38](#_Toc87026597)

[5.12.6 Three Weaknesses: 38](#_Toc87026598)

[5.12.7 Dry Run: 38](#_Toc87026599)

[5.13 Patience Sort: 39](#_Toc87026600)

[5.13.1 Description: 39](#_Toc87026601)

[5.13.2 Pseudo Code: 39](#_Toc87026602)

[5.13.3 Code in Python: 40](#_Toc87026603)

[5.13.4 Time Complexity Analysis: 41](#_Toc87026604)

[5.13.5 Three Strengths: 42](#_Toc87026605)

[5.13.6 Three Weaknesses: 42](#_Toc87026606)

[5.13.7 Dry Run: 42](#_Toc87026607)

[6 Multi-level Sorting: 42](#_Toc87026608)

[6.1 Code in Python: 43](#_Toc87026609)

[7 Searching Algorithms: 44](#_Toc87026610)

[7.1 Linear Search: 44](#_Toc87026611)

[7.1.1 Description: 44](#_Toc87026612)

[7.1.2 Pseudo Code: 44](#_Toc87026613)

[7.1.3 Code in Python: 44](#_Toc87026614)

[7.1.4 Time Complexity Analysis: 44](#_Toc87026615)

[7.1.5 Dry run: 44](#_Toc87026616)

[7.2 Binary Search: 44](#_Toc87026617)

[7.2.1 Description: 44](#_Toc87026618)

[7.2.2 Pseudo Code: 45](#_Toc87026619)

[7.2.3 Code in Python: 45](#_Toc87026620)

[7.2.4 Time Complexity Analysis: 45](#_Toc87026621)

[7.2.5 Dry run: 46](#_Toc87026622)

[7.3 Jump Search: 46](#_Toc87026623)

[7.3.1 Description: 46](#_Toc87026624)

[7.3.2 Pseudo Code: 46](#_Toc87026625)

[7.3.3 Code in Python: 46](#_Toc87026626)

[7.3.4 Time Complexity Analysis: 47](#_Toc87026627)

[7.3.5 Dry Run: 47](#_Toc87026628)

[7.4 Exponential Search: 47](#_Toc87026629)

[7.4.1 Description: 47](#_Toc87026630)

[7.4.2 Pseudo Code: 47](#_Toc87026631)

[7.4.3 Code in Python: 47](#_Toc87026632)

[7.4.4 Time Complexity Analysis: 48](#_Toc87026633)

[7.4.5 Dry Run: 48](#_Toc87026634)

[8 Composite Searching: 48](#_Toc87026635)

[8.1 Code in Python: 48](#_Toc87026636)

[9 UI: 50](#_Toc87026637)

[9.1 Wireframes: 50](#_Toc87026638)

[9.1.1 Start Window: 50](#_Toc87026639)

[9.1.2 Scrap Window: 50](#_Toc87026640)

[9.1.3 Sort Window: 51](#_Toc87026641)

[9.1.4 Search Window: 51](#_Toc87026642)

[9.2 Implementation: 52](#_Toc87026643)

[9.2.1 Start Window: 52](#_Toc87026644)

[9.2.2 Scrap Window: 53](#_Toc87026645)

[9.2.3 Sort Window: 54](#_Toc87026646)

[9.2.4 Search Window: 55](#_Toc87026647)

[10 Integration: 56](#_Toc87026648)

[10.1 Class Diagrams: 56](#_Toc87026649)

[10.1.1 Scrapping.py 56](#_Toc87026650)

[10.1.2 Start\_Window\_UI.py 56](#_Toc87026651)

[10.1.3 Scrap\_Window\_UI.py 57](#_Toc87026652)

[10.1.4 Sort\_Window\_UI.py 57](#_Toc87026653)

[10.1.5 Search\_Window\_UI.py 58](#_Toc87026654)

[10.1.6 Start\_Window\_UI\_Controller.py 58](#_Toc87026655)

[10.1.7 Scarp\_Window\_UI\_Controller.py 59](#_Toc87026656)

[10.1.8 Sort\_Window\_UI\_Controller.py 59](#_Toc87026657)

[10.1.9 Search\_Window\_UI\_Controller.py 60](#_Toc87026658)

[10.1.10 Sort.py 60](#_Toc87026659)

[10.1.11 Insertion\_Sort.py 60](#_Toc87026660)

[10.1.12 Selection\_Sort.py 61](#_Toc87026661)

[10.1.13 Radix\_Sort.py 61](#_Toc87026662)

[10.1.14 Quick\_Sort.py 61](#_Toc87026663)

[10.1.15 Patience\_Sort.py 62](#_Toc87026664)

[10.1.16 Pancake\_Sort.py 62](#_Toc87026665)

[10.1.17 Merge\_Sort.py 62](#_Toc87026666)

[10.1.18 Heap\_Sort.py 63](#_Toc87026667)

[10.1.19 Cycle\_Sort.py 63](#_Toc87026668)

[10.1.20 Count\_Sort.py 63](#_Toc87026669)

[10.1.21 Bucket\_Sort.py 63](#_Toc87026670)

[10.1.22 Bubble\_Sort.py 64](#_Toc87026671)

[10.1.23 Bitonic\_Sort.py 64](#_Toc87026672)

[10.1.24 Multi-level\_Sorting.py 64](#_Toc87026673)

[10.1.25 Search.py 64](#_Toc87026674)

[10.1.26 Linear\_Search.py 65](#_Toc87026675)

[10.1.27 Binary\_Search.py 65](#_Toc87026676)

[10.1.28 Exponential\_Search.py 65](#_Toc87026677)

[10.1.29 Jump\_Search.py 65](#_Toc87026678)

[10.1.30 Filter\_Data.py 65](#_Toc87026679)

[10.1.31 Composite\_Filter.py 66](#_Toc87026680)

[10.1.32 Time.py 66](#_Toc87026681)

[10.1.33 Random.py 66](#_Toc87026682)

[10.1.34 TxtFile\_Handling.py 66](#_Toc87026683)

[11 Collaboration: 67](#_Toc87026684)

# Description:

This project is based on the data retrieval system in which about 1 million data about books would be scrapped. Some attributes of books will be randomly generated on the run time. In the attributes of each book its name, id, price, number of pages, author name, language and type will be included. While data is being scrapped it can be stopped or paused. And the website from which data is being scrapped, it’s URL will also be displayed on the screen. As the data is very large and it will become difficult to scrap data from a single website, we will be using multiple websites or multiple urls from single website but different categories. to collect large data. When the data is retrieved it can be sorted using multiple algorithms and it can be sorted in any order decreasing or increasing. Multi-level sorting will also be implement on multiple columns at the same time. After sorting of column, time taken to sort that data will also be displayed in milliseconds. User will also be able to search from multiple columns based on multiple searching algorithms and filters. Data would also be searched using composite filters such as and, or and not. User will also have option to choose any searching algorithm or filter. We will use pyqt framework to implement our project in GUI and the design will be implemented in QT Designer software.

# Motivation:

We choose this project to know that how actually the large data is managed within less time using efficient algorithms.

# Target Audience:

Librarian will be the end user of this product.

# Business Need:

We're a designing a software that will help a librarian to manage the library efficiently. We are developing a software that will help librarians of a large library with up to 1 million books. He/she can easily check the availability of books. The file will contain the complete sorted data of all library books so the librarian can manage their tasks in time and they don't have to perform it manually and face the hard work.

# Data (Scrapping):

We were required a large number of data to apply sorting and searching algorithms in an efficient way. So, we found a [website](https://archive.org/) from where we scrapped about 0.8 Million data of books. We scrapped 7 attributes of each book i.e. Name, Price, Id, Pages, Author, Language and Type. We spent continuous 10+ hours to scrape that data. We were able to get 75 books data from a single webpage. And in each category almost 10,000. We saved all the categories in a text file.

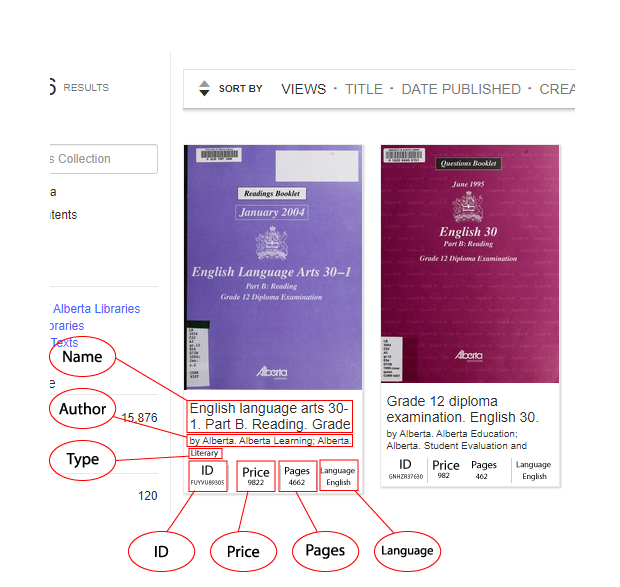


Figure 1: Scrapping example page



Figure 2: Scrapping URL

# Sorting Algorithms:

## Selection Sort:

### Description:

It is a simple algorithm. In this algorithm the smallest element in the whole array is selected and moved to the first. The process is repeated for the whole array until the whole array is not sorted. The time complexity of this algorithm is O(n2).

### Pseudo Code:

for I = 0 to arr.length()  
 for j = I + 1 to arr.length()  
 if arr[i] > arr[j]  
 swap arr[i], arr[j]

### Code in Python:

# Implementation of Selection Sort for both Numbers and Strigs

class SelectionSort(Sort):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    # method to sort numbers

    def sort(self, reverse=False):

        for i in range(self.length):

            for j in range(i + 1, self.length):

                if (self.arr[i] > self.arr[j] and reverse == False):

                    self.arr[i], self.arr[j] = self.arr[j], self.arr[i]

                elif (self.arr[i] < self.arr[j] and reverse == True):

                    self.arr[i], self.arr[j] = self.arr[j], self.arr[i]

        return self.arr

### Time Complexity Analysis:

for I = 0 to arr.length() n  
 for j = I + 1 to arr.length() n(n – 1)  
 if arr[i] > arr[j] n(n – 2)  
 swap arr[i], arr[j] n(n – 2)

### Strengths:

It runs better on small number of elements.

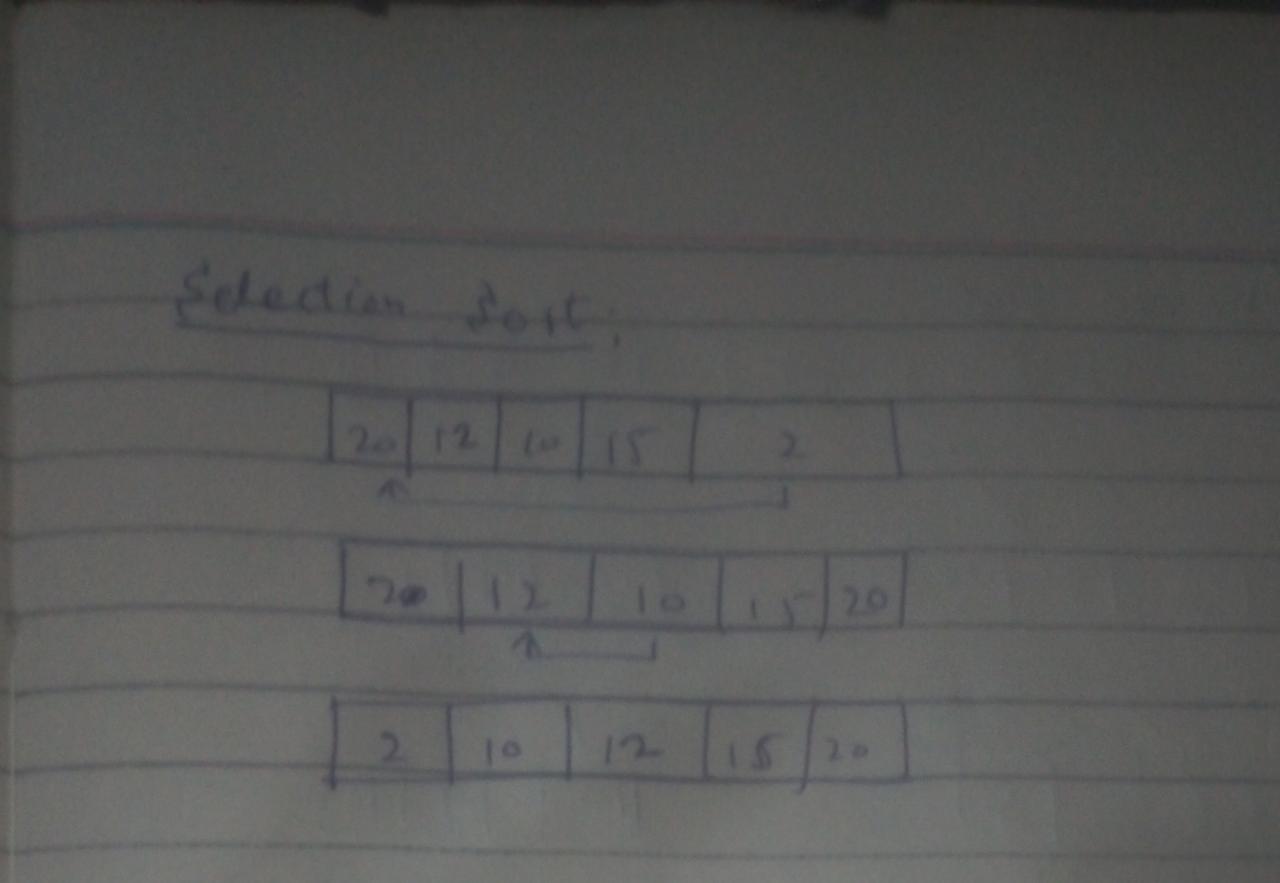
No extra space is required for sorting the elements.

### Weakness:

It runs slowly on large number of elements.

It requires n ^ n time to sort elements.

### Dry run on small input:



## Insertion sort:

### Description:

In this algorithm elements of array are compared and swapped. It starts from 2nd index of the array and if it is larger than the element in its left then it remains on its position otherwise it is swapped with the left element. In the next iterations the process is repeated for all the elements in the array until the whole array is not sorted. Its time complexity is O(n2).

### Pseudo Code:

for i = 0 to A.length  
 for j = i to 0  
 if A[j] < A[j – 1]  
 swap A[j], A[j – 1]

return A

### Code in Python:

# Implementation of Insertion Sort for both Numbers and Strigs

class InsertionSort(Sort):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    def sort(self, reverse=False):

        for i in range(1, self.length):

            for j in range(i, 0, -1):

                if(self.arr[j] < self.arr[j - 1] and reverse == False):

                    self.arr[j], self.arr[j - 1] = self.arr[j - 1],  self.arr[j]

                elif(self.arr[j] > self.arr[j - 1] and reverse == True):

                    self.arr[j], self.arr[j - 1] = self.arr[j - 1],  self.arr[j]

                else:

                    break

        return self.arr

### Time Complexity Analysis:

for i = 0 to A.length n  
 for j = i to 0 n(n – 1)  
 if A[j] < A[j – 1] n(n – 2)  
 swap A[j], A[j – 1] n(n – 2)

return A

### Strengths:

It runs better on small number of elements.

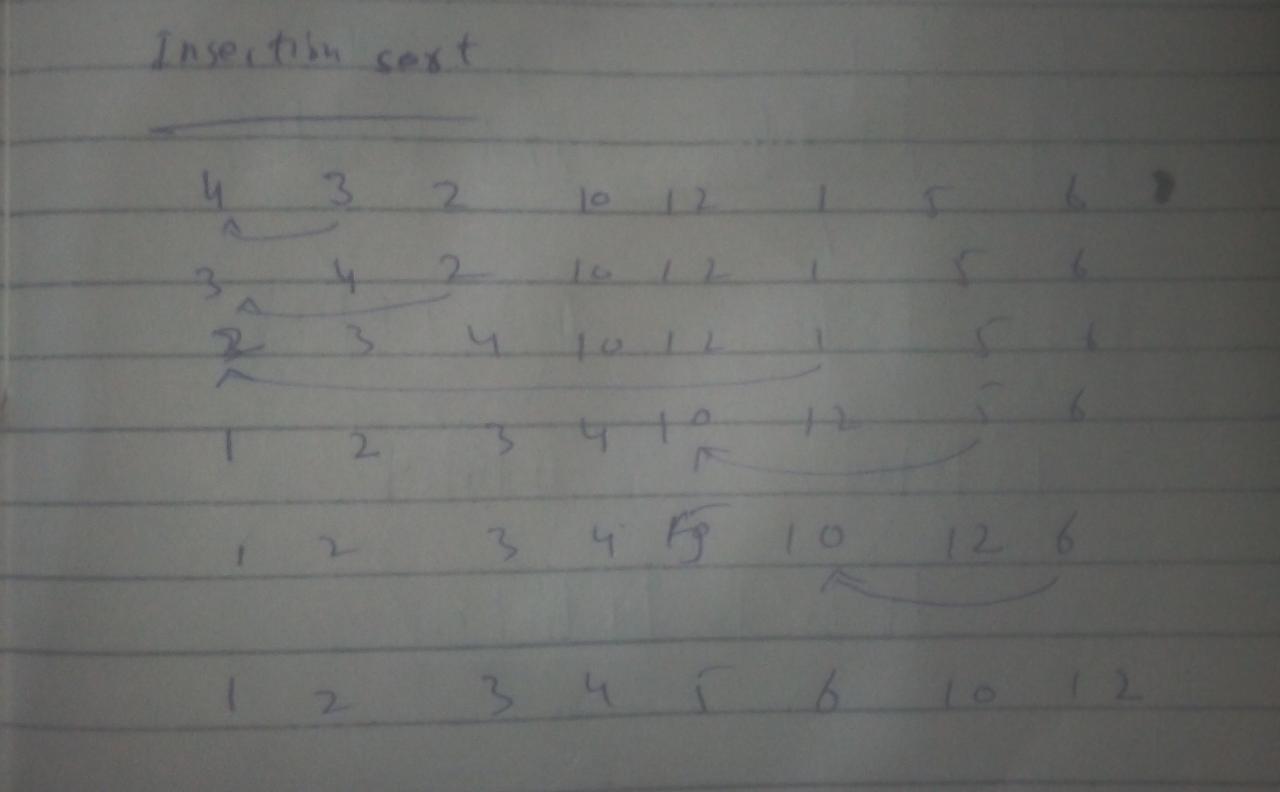
No extra space is required for sorting the elements.

### Weakness:

It runs slowly on large number of elements.

It requires n ^ n time to sort elements.

### Dry run on small input:



## Bubble Sort:

In this algorithm elements of the array are compared and swapped and moved to the end of the list until no more possible swaps are left. Its time complexity is O(n2).

### Pseudo Code:

for I = 0 to A.length  
 swapped <= False  
 for j = 0 to A.Length – 1  
 if A[j] > A[j + 1]  
 swap A[j], A[j + 1]

return A

### Code in Python:

class BubbleSort(Sort):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    def sort(self, reverse=False):

        for j in range(self.length):

            swapped = False

            for i in range(self.length - 1):

                # for ascending order sort

                if (self.arr[i] > self.arr[i + 1] and reverse == False):

                    self.arr[i], self.arr[i + 1] = self.arr[i + 1], self.arr[i]

                    swapped = True

                # for descending order sort

                elif (self.arr[i] < self.arr[i + 1] and reverse == True):

                    self.arr[i], self.arr[i + 1] = self.arr[i + 1], self.arr[i]

                    swapped = True

            if swapped == False:

                break

        return self.arr

### Time Complexity Analysis:

for I = 0 to A.length n  
 swapped <= False n- 1  
 for j = 0 to A.Length – 1 n(n – 1)  
 if A[j] > A[j + 1] n(n – 2)  
 swap A[j], A[j + 1] n(n – 2)

return A

### Strengths:

It runs better on small number of elements.

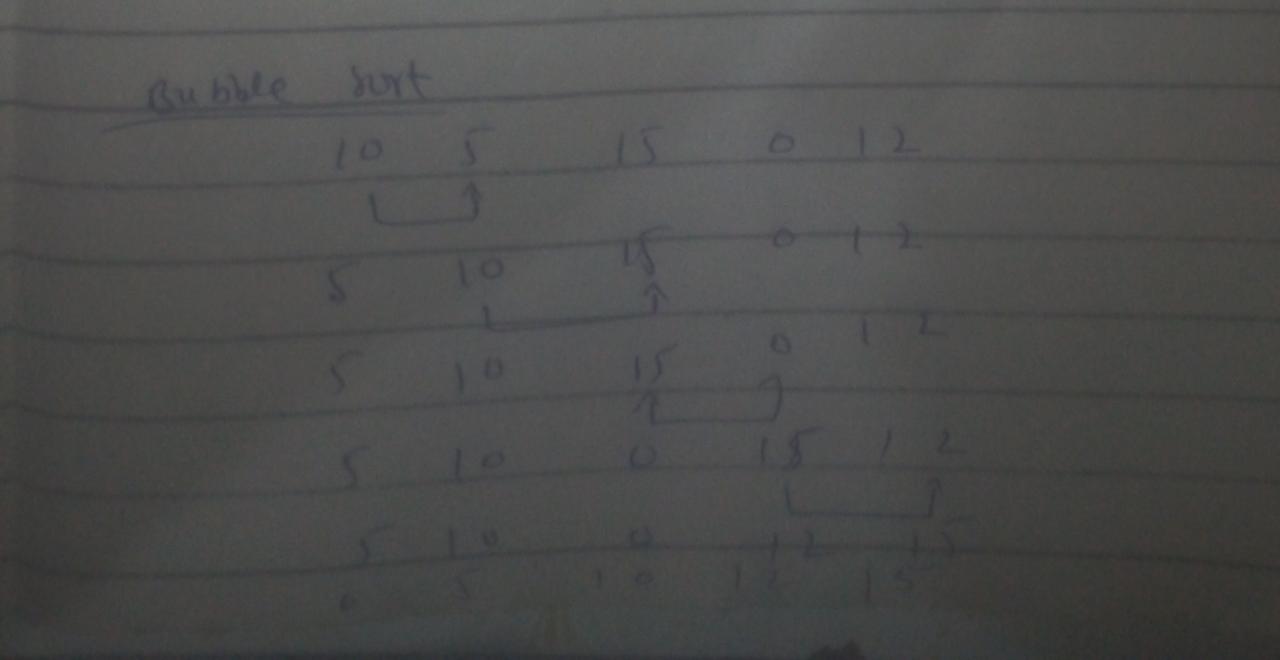
No extra space is required for sorting the elements.

### Weakness:

It runs slowly on large number of elements.

It requires n ^ n time to sort elements.

### Dry run on small input:



## Merge Sort:

### Description:

It is a recursive algorithm. In this algorithm the array is divided into to subarrays and the process is recursively repeated until the sub array is reduced to only one element. Then the two divided subarrays are sorted using merge method.

### Pseudo Code:

Merge(A, p, q, r)

n1 = q – p + 1  
n2 = r – q  
let L [1..n1 + 1] and R[1.. n2 + 1] be new arrays  
for i = 1 to n1  
 L[i] = A[p + I – 1]  
for j = 1 to n2  
 R[j] = A[q + j]  
L[n1 + 1] = infinity  
R[n2 + 1] = infinity  
i = 1  
j = 1  
for k = p to r  
 if L[i] <= R[j]  
 A[k] = L[i]  
 i += 1  
 else A[k] = R[j]  
 j = j + 1

MERGE-SORT (A, p, r)  
 if p < r  
 q = (p + r) / 2  
 MERGE-SORT (A, p, q)  
 MERGE-SORT (A, q + 1, r)  
 MERGE(A, p, q, r)

### Code in Python:

class MergeSort(Sort):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    # sort

    def sort(self, reverse=False):

        self.mergeSort(0, len(self.arr) - 1, reverse)

        return self.arr

    # merge

    def merge(self, left, mid, right, reverse):

        L = []

        R = []

        for i in range(left, mid + 1):

            L.append(self.arr[i])

        for i in range(mid + 1, right + 1):

            R.append(self.arr[i])

        if reverse == False and type(self.arr[0]) == int:

            L.append(100000000000000000)

            R.append(100000000000000000)

        elif reverse == True and type(self.arr[0]) == int:

            L.append(-100000000000000000)

            R.append(-100000000000000000)

        elif reverse == False and type(self.arr[0]) == str:

            L.append("z")

            R.append("z")

        elif reverse == True and type(self.arr[0]) == str:

            L.append("A")

            R.append("A")

        i = 0

        j = 0

        for k in range(left, right+1):

            # for ascending order

            if reverse == False:

                if L[i] <= R[j]:

                    self.arr[k] = L[i]

                    i += 1

                else:

                    self.arr[k] = R[j]

                    j += 1

            # for descending order

            else:

                if L[i] >= R[j]:

                    self.arr[k] = L[i]

                    i += 1

                else:

                    self.arr[k] = R[j]

                    j += 1

    # merge sort

    def mergeSort(self, left, right, reverse):

        if left < right:

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

            self.mergeSort(left, mid, reverse)

            self.mergeSort(mid + 1, right, reverse)

            self.merge(left, mid, right, reverse)

### Time Complexity Analysis:

MERGE-SORT (A, p, r)  
 if p < r lgn  
 q = (p + r) / 2 lgn  
 MERGE-SORT (A, p, q) lgn  
 MERGE-SORT (A, q + 1, r) lgn  
 MERGE(A, p, q, r) nlgn

### Strengths:

It works better on large number on elements.

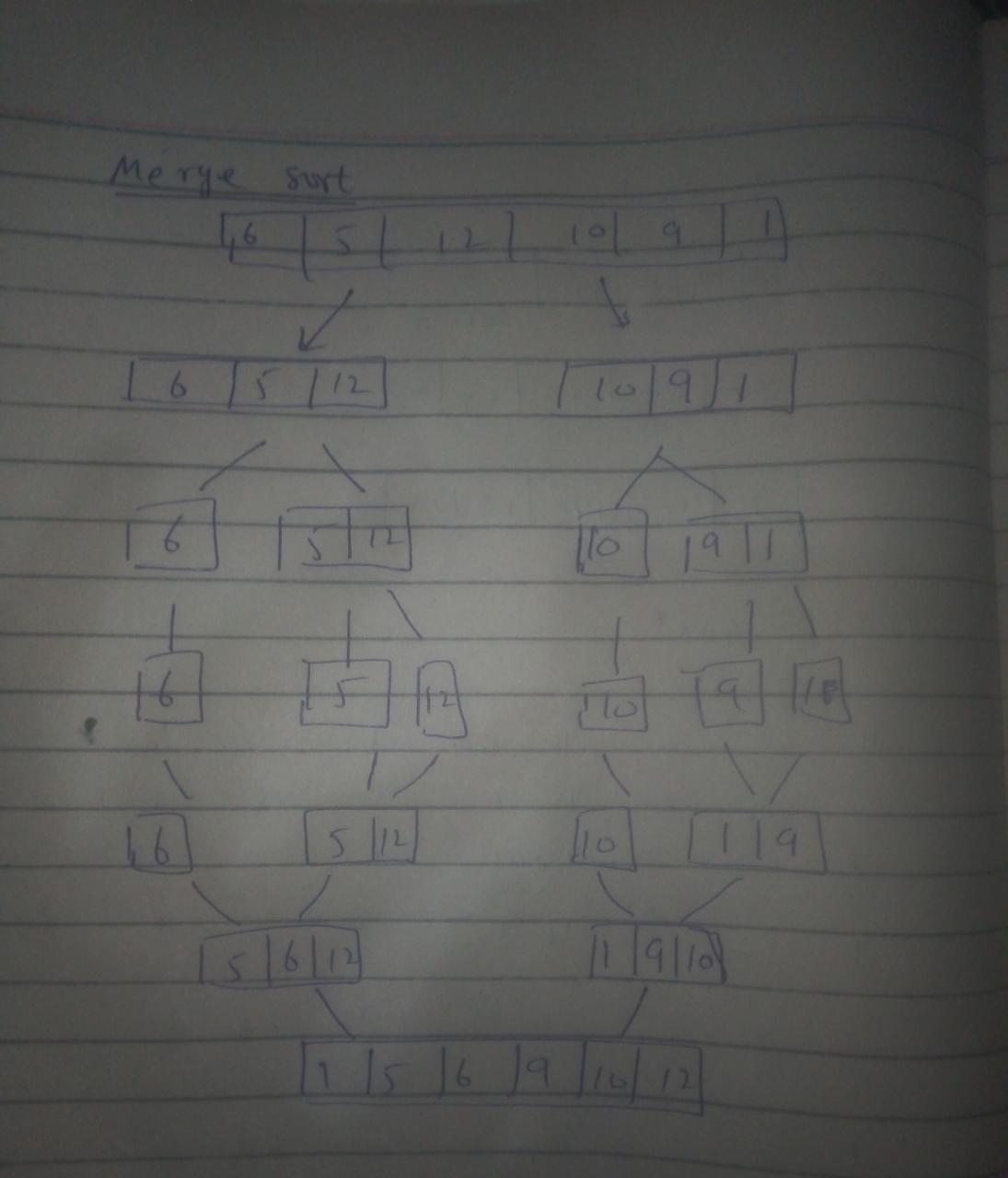
It’s time complexity is nlgn which is faster than n ^ n

### Weakness:

It runs slowly for small number of elements.

It uses extra spaces to create sub arrays

### Dry run on small input:



## Cycle Sort:

### Description:

It involves cycles to sort whole array. In this we start from 0 index select that element from array and check from whole array that if the selected array index value is greater than other array index we kept on moving in checking the array until the selected array index value is less than the specific value of array element we first save that array index value and then we place already selected value over that array index this process continues until the whole array is sorted.

### Pseudo Code:

for count = 0 to A.length  
 item = A[count]  
 counter = count  
 for i = counter + 1 to A.length  
 if A[i] < item  
 counter += 1  
 swap item, A[counter]  
  
 while counter != count  
 counter = count  
 for i = counter + 1 to A.length  
 if A[i] < item  
 counter += 1  
 swap item, A[counter]

### Code in Python:

# Implementation of Cycle Sort for both Numbers and Strigs

class CycleSort(Sort):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    def sort(self, reverse=False):

        for c in range(self.length):

            item = self.arr[c]

            counter = c

            for i in range(c + 1, self.length):

                if self.arr[i] < item:

                    counter += 1

            item, self.arr[counter] = self.arr[counter], item

            while(counter != c):

                counter = c

                for i in range(c + 1, self.length):

                    if self.arr[i] < item:

                        counter += 1

                item, self.arr[counter] = self.arr[counter], item

        if reverse:

            return self.arr[::-1]

        return self.arr

### Proof of correctness:

for count = 0 to A.length n  
 item = A[count] n - 1  
 counter = count n - 1  
 for i = counter + 1 to A.length n(n -1)  
 if A[i] < item n(n – 2)  
 counter += 1 n(n -1)  
 swap item, A[counter] n - 1  
  
 while counter != count n( n -1)  
 counter = count n ( n -2)  
 for i = counter + 1 to A.length n( n -1)(n – 1)  
 if A[i] < item n(n-1)(n-2)  
 counter += 1 n(n-1)(n-2)  
 swap item, A[counter] n(n-2)

### Strengths:

It uses less number of memory writes.

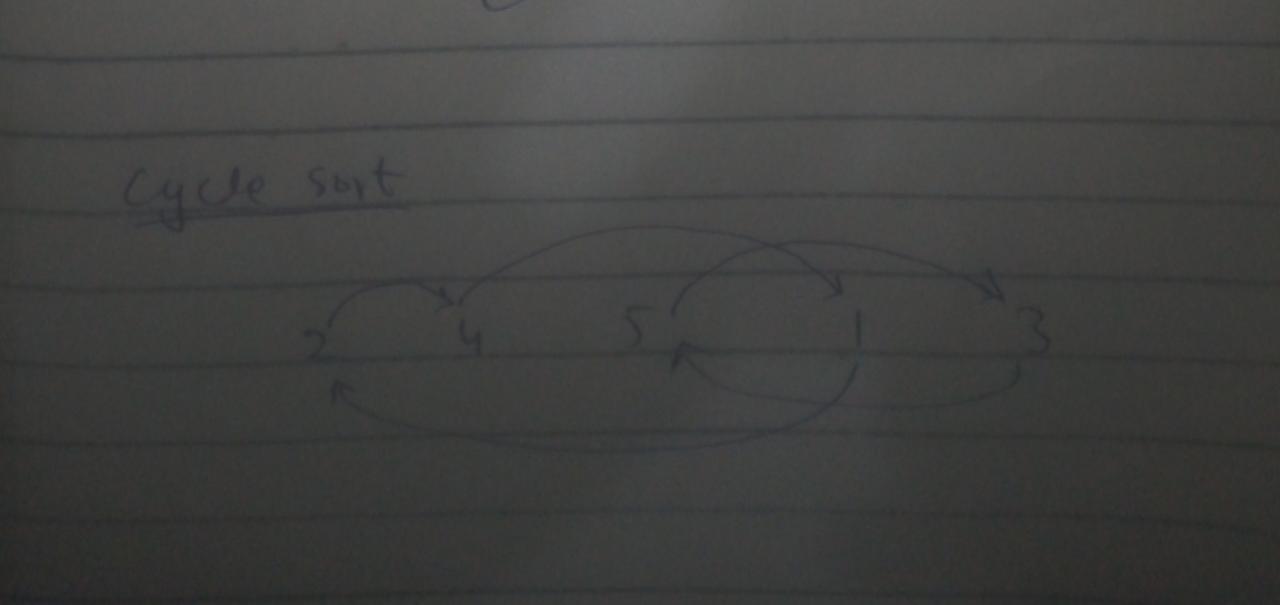
No extra space is required for sorting the elements.

### Weakness:

It forces to divide data in cycles

Due to time complexity of more than n ^ n it is very slow

Dry run on small input:



## Bitonic Sort:

### Description:

It first divides array into two parts consider in left, right arrays and sort left array in increasing order and right array in decreasing order after that both left, right arrays are compared smaller elements are swapped in first half (left array) and greater elements in second half (right array) and then we merge both left, right arrays to get final sorted array.

### Pseudo Code:

compare(i, j, direction)  
 if direction == 1 and A[i] > A[j] or direction == 1 and A[i] < A[j]  
 swap A[i], A[j]  
merge(a, b, direction)  
if b > 1:  
 k = b / 2  
 for i = a to a + k  
 compare(i, i+k, direction)  
 merge(a, k, direction)  
 merge(a + k, k, direction)  
sort(a, b, direction)  
 if b > 1  
 k = b / 2  
 sort(a, k, 1)  
 sort(a + k, k, 0)  
 merge(a, b, direction)

### Code in Python:

# Implementation of Bitonic Sort for both Numbers and Strigs

class BitonicSort(Sort):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    def sort(self, reverse=False):

        self.sortBitonic(0, self.length, 1)

        if reverse:

            return self.arr[::-1]

        return self.arr

    def compare(self, i, j, direction):

        if (direction == 1 and self.arr[i] > self.arr[j]) or (direction == 0 and self.arr[i] < self.arr[j]):

            self.arr[i], self.arr[j] = self.arr[j], self.arr[i]

    def merge(self, a, b, direction):

        if b > 1:

            k = int(b/2)

            for i in range(a, a+k):

                self.compare(i, i+k, direction)

            self.merge(a, k, direction)

            self.merge(a+k, k, direction)

    def sortBitonic(self, a, b, direction):

        if b > 1:

            k = int(b/2)

            self.sortBitonic(a, k, 1)

            self.sortBitonic(a+k, k, 0)

            self.merge(a, b, direction)

### Time Complexity Analysis:

sort(a, b, direction)  
 if b > 1 lgn  
 k = b / 2 lgn  
 sort(a, k, 1) lgn  
 sort(a + k, k, 0) lgn  
 merge(a, b, direction) nlgn

### Three strengths:

It works better on large number on elements.

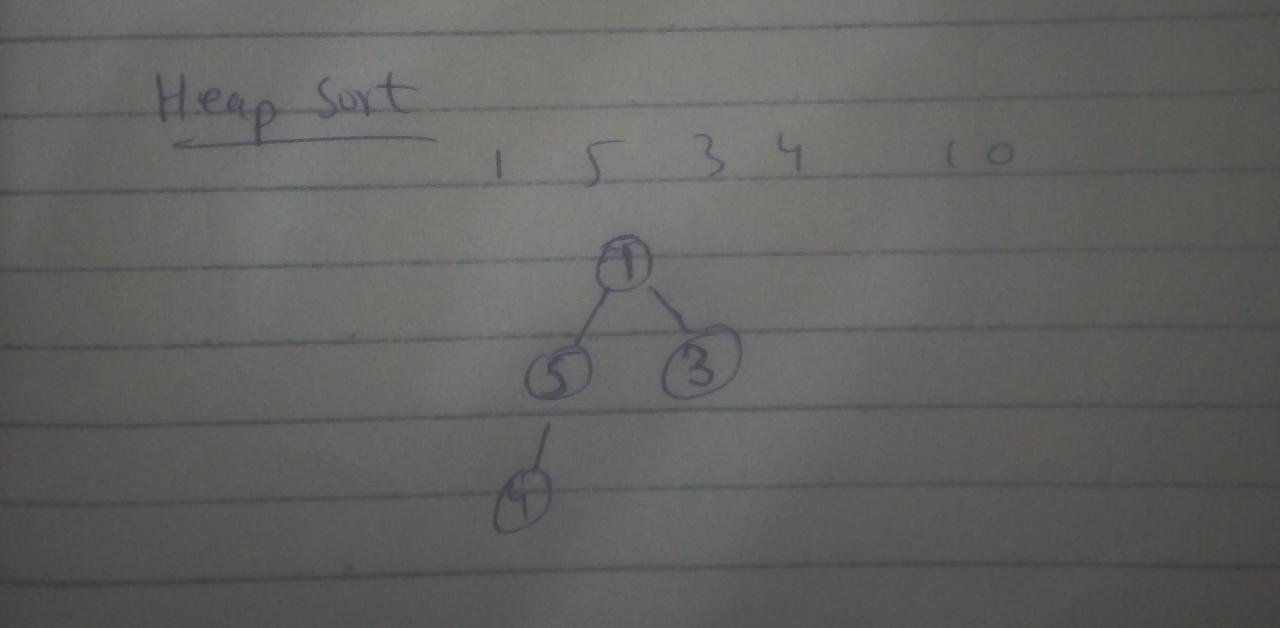
It’s time complexity is nlgn which is faster than n ^ n

### Three weakness:

It runs slowly for small number of elements.

It uses extra spaces to create sub arrays

### Dry run on small input:



## Heap Sort:

Heap sort is also a recursive algorithm in which a binary tree structure is created. Like selection sort in this algorithm a smallest element is placed at the top of the heap and this element is removed and added to a new list and the process is repeated until the whole list is sorted.

### Pseudo Code:

for I = A.length -1 to 0  
 makeHeap(A.length, i)  
for j = A.length – 1, 0  
 swap A[j], A[0]  
 makeHeap(j, 0)  
makeHeap(n, i)  
 largest = i  
 left = 2 \* i + 1  
 rigth = 2 \* i + 2  
 if left < n and A[largest] < A[left]  
 largest = left  
 if right < n and A[largest] < A[right]  
 largest = right  
 if largest != i  
 swap A[i], A[largest]

### Code in Python:

class HeapSort(Sort):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    def sort(self, reverse=False):

        for i in range((self.length // 2) - 1, -1, -1):

            self.makeHeaf(self.length, i)

        for j in range(self.length - 1, 0, -1):

            self.arr[j], self.arr[0] = self.arr[0], self.arr[j]

            self.makeHeaf(j, 0)

        return self.arr

    def makeHeaf(self, n, i):

        largest = i

        left = 2 \* i + 1

        right = 2 \* i + 2

        if left < n and self.arr[largest] < self.arr[left]:

            largest = left

        if right < n and self.arr[largest] < self.arr[right]:

            largest = right

        if largest != i:

            self.arr[i], self.arr[largest] = self.arr[largest], self.arr[i]

            self.makeHeaf(n, largest)

### Time Complexity Analysis:

for I = A.length -1 to 0 n  
 makeHeap(A.length, i) n - 1  
for j = A.length – 1, 0 n  
 swap A[j], A[0] n - 1  
 makeHeap(j, 0) n - 1

### Three strengths:

Heap Sort Algorithm is a very efficient algorithm.

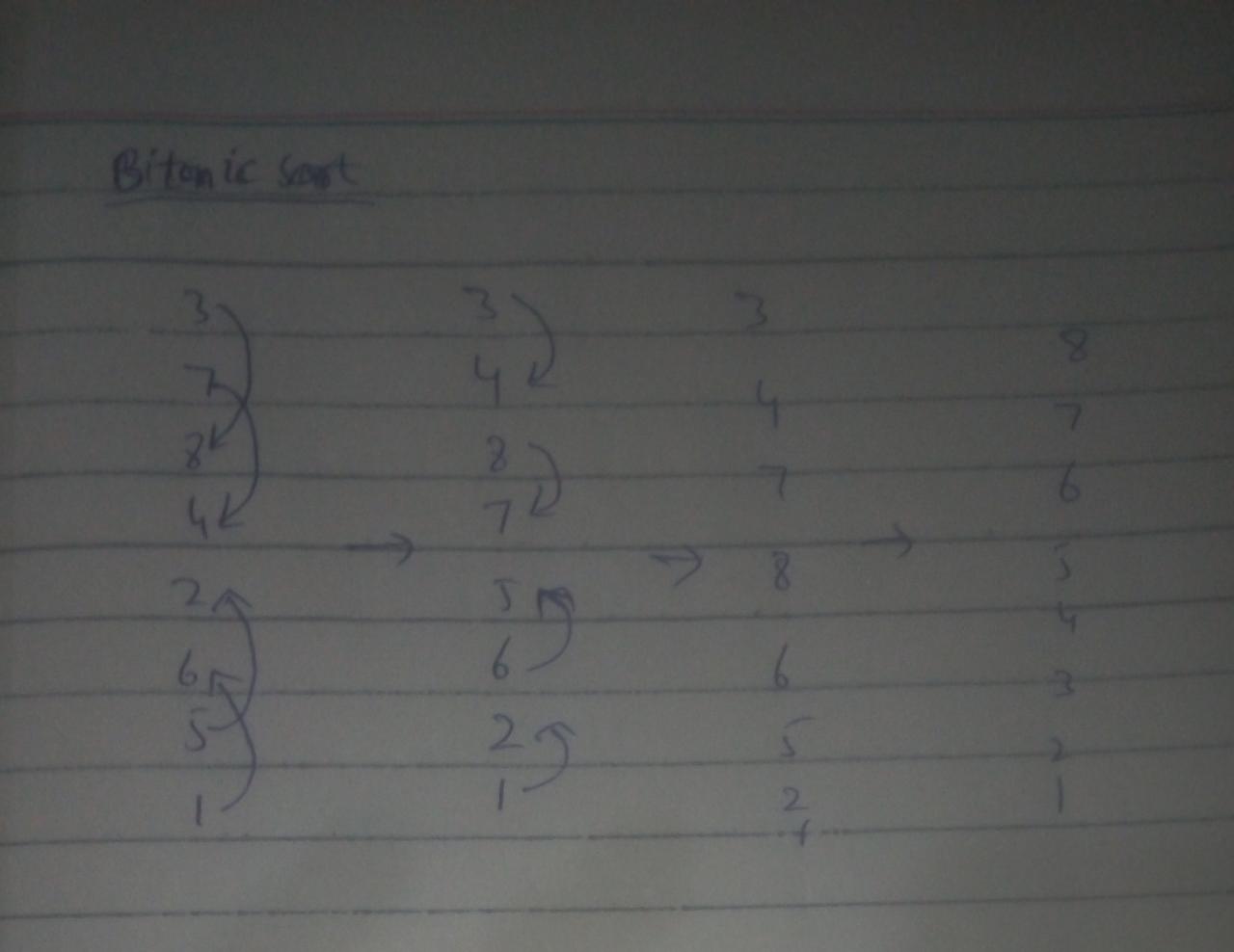
It’s time complexity is nlgn which is faster than n ^ n

### Three weakness:

This algorithm takes more time to sort data.

In this algorithm memory management is complex.

### Dry run on small input:



## Bucket Sort:

### Description:

It is a sorting algorithm that works by distributing the elements of an array into a number of buckets. It puts each number from array into its respective bucket in which it should be. Each bucket is then sorted individually by applying any sorting algorithm (Like insertion sort etc.). Finally, the sorted buckets are combined to form a final sorted array. Its time complexity in best/average case is O(n+k) and in worst case is O(n2).

### Pseudo Code:

for I = 0 to 10

arr 🡨 append([])  
for j = 0 to x

arr[10 x j] 🡨 append(j)

for s = 0 to arr

insertionSort(s)

for i in arr

for j in i

x[counter] 🡨 j

counter 🡨 counter + 1

### Code in Python:

# Implementation of Bucket Sort for Floating Point Numbers

class BucketSort(sort):

    def \_\_init\_\_(self, arr):

        self.arr = arr

    def sortInt(self, reverse=False):

        return self.bucketSort(self.arr,reverse)

    def insertionSort(self,array):

        for j in range(0, len(array)):

            key = array[j]

            i = j-1

            while(i >= 0 and array[i] > key):

                array[i+1] = array[i]

                i = i-1

            array[i+1] = key

        return (array)

    def bucketSort(self,x,reverse):

        arr = []

        buckets = 10

        for i in range(0,buckets):

            arr.append([])

        for i in x:

            ind = int(10 \* i)

            arr[ind].append(i)

        for i in arr:

            self.insertionSort(i)

        counter = 0

        for i in arr:

            for j in i:

                x[counter] = j

                counter += 1

        if(reverse == True):

            return x[:: -1]

        else:

            return x

### Time Complexity Analysis:

for I = 0 to 10 (n)

arr 🡨 append([]) (n-1)  
for j = 0 to x (n)

arr[10 x j] 🡨 append(j) (n-1)

for s = 0 to arr (n)

insertionSort(s) (n-1)

for i in arr (n)

for j in i (n(n-1))

x[counter] 🡨 j (n-2)

counter 🡨 counter + 1 (n-2)

### Three Strengths:

In Bucket Sort we are able to sort each bucket separately in this way main large array is divided into smaller array and we can sort smaller arrays quickly by using insertion sort algorithm which works fast on array with smaller elements.

Bucket Sort also helps us to fit large list into memory because as we said earlier it divides a large list into smaller buckets.

Bucket Sort is a stable sort and works fast for equally divided list fast in the case if we want bucket sort to apply on non stable situation we use here Quick Sort to sort buckets but in a stable situation we use merge sort or insertion sort.

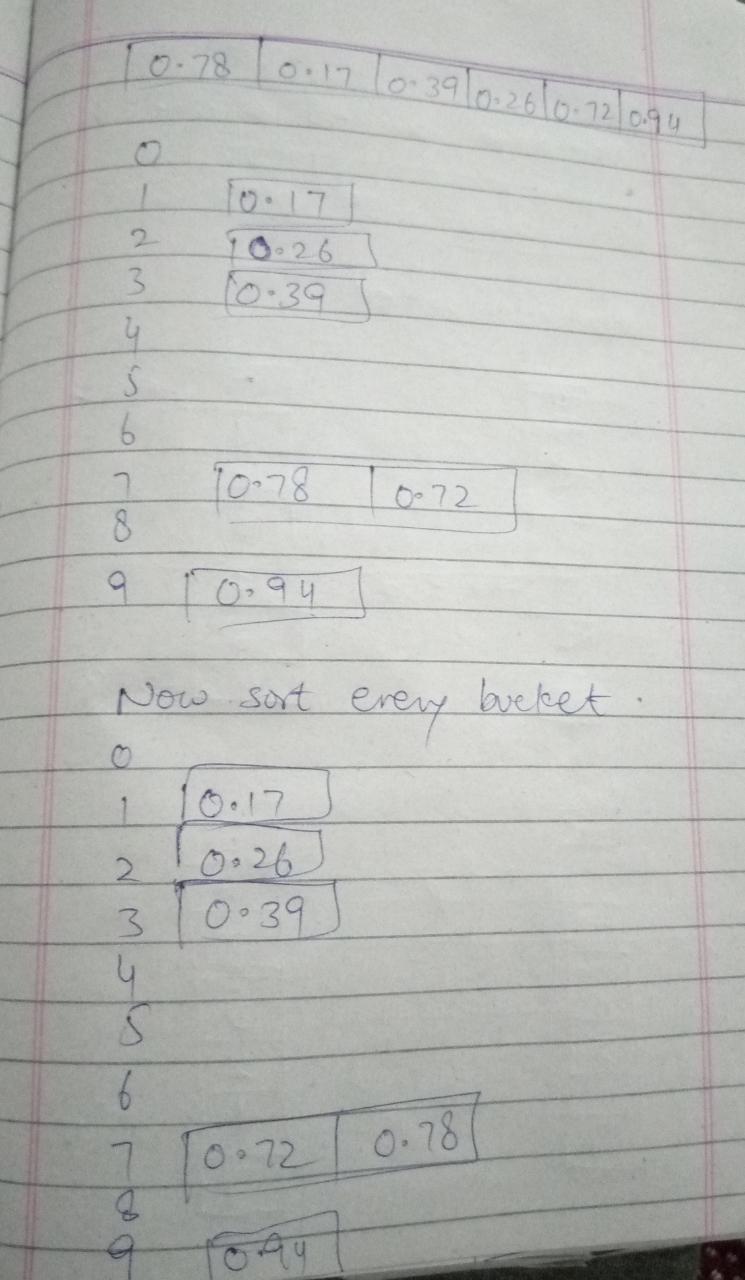
### Three Weaknesses:

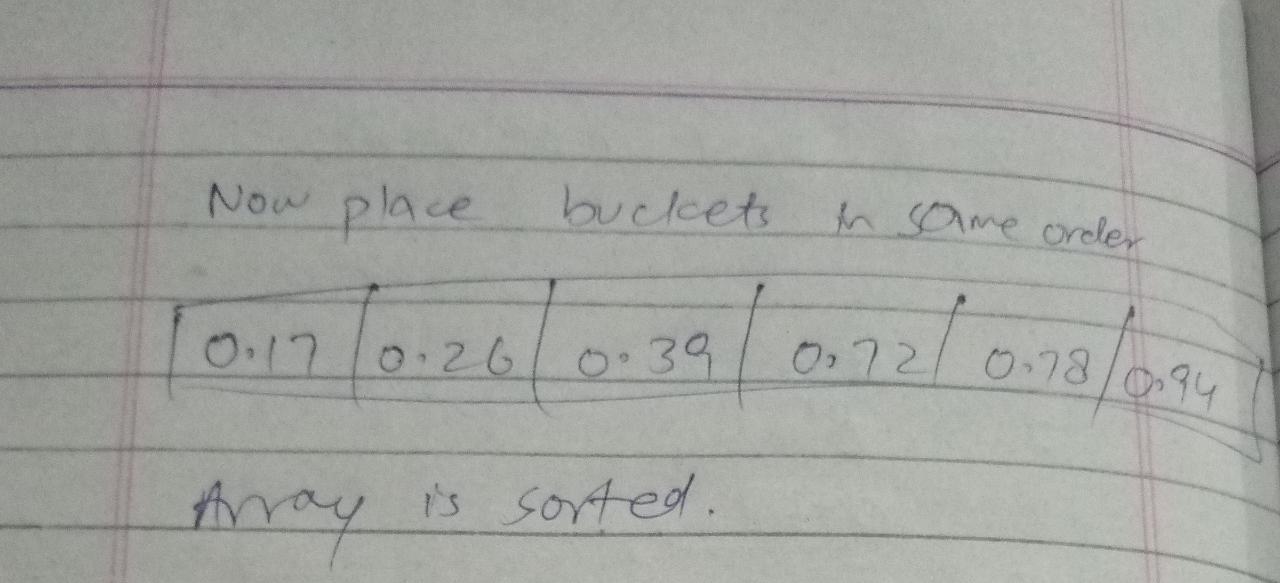
In Bucket sort if the buckets are not equally divide we apply some extra effort because as in its algorithm we first pick last element of each bucket and then compare it with every last element of each bucket if it is smaller then we put it in array first the problem is that is buckets are not equal then it will take extra efforts to check last element of each bucket and place it in new array.

Bucket Sort is only applicable on floating point values we can’t apply it on strings or integers because it’s difficult to make buckets in that case.

If buckets will be large then it will be take more time to apply insertion sort on buckets to sort them.

### Dry Run:





## Radix Sort:

### Description:

In this, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from least significant digit to the most significant digit. It will be completed in (n) number of steps where n is equal to number of digits of largest number in array. For example, if the largest number is a 300 so it has 3 digits in it then list is sorted with 3 numbers of steps. It will place the numbers being sorted on the basis of starting from least significant digit and then onwards until most significant digit in their respective buckets in the end we will place the buckets in main array and we will be able to see sorted array. Its time complexity is O(n).

### Pseudo Code:

while((Max // pos) > 0)

Use unstable sort(like CountSort etc..) to sort Array

pos 🡨 pos x 10

### Code in Python:

# Implementation of Radix Sort for Integers

class radixSort(sort):

    def \_\_init\_\_(self, arr,pos):

        self.arr = arr

        self.pos = pos

    def sortInt(self, reverse=False):

        return self.RadixSort(self.arr,reverse)

    def RadixSort(self,arr,reverse):

        Max = max(arr)

        pos = 1

        while((Max // pos) > 0):

            self.countSort(self.arr, pos)

            pos \*= 10

        if(reverse == True):

            return self.arr[:: -1]

        else:

            return self.arr

    def countSort(self,arr, pos):

        output = [0] \* (len(self.arr))

        count = [0] \* (10)

        for i in range(0, len(arr)):

            index = (self.arr[i] // pos)

            count[index % 10] += 1

        for i in range(1, 10):

            count[i] += count[i - 1]

        i = len(self.arr) - 1

        while i >= 0:

            value = (self.arr[i] // pos)

            output[count[value % 10] - 1] = self.arr[i]

            count[value % 10] -= 1

            i -= 1

        for i in range(0, len(self.arr)):

            self.arr[i] = output[i]

### Time Complexity Analysis:

while((Max // pos) > 0) (n)

Use unstable sort(like CountSort etc...) to sort Array (n-1)

pos 🡨 pos x 10 (n-1)

### Three Strengths:

Radix Sort works more efficiently when there are integers with fewer and fixed number of digits.

Radix Sort has linear time complexity.

Radix Sort numbers digit by digit and the digit range is fixed which 0-9 which is an advantage.

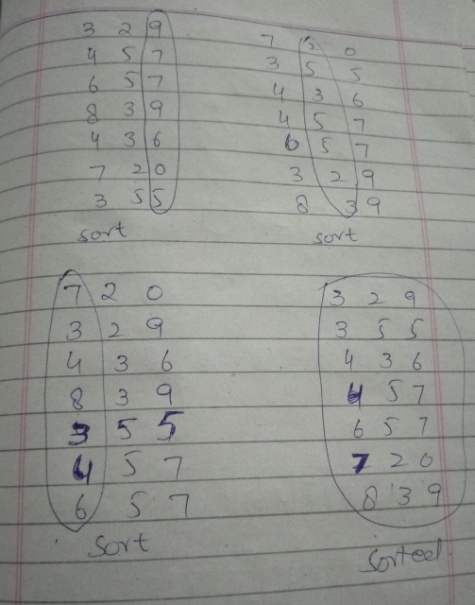
### Three Weaknesses:

Radix sort only works on that numbers with fixed number if digits if the number is too large it will become slow because radix sort itself use count sort and it will make a large array which will slow down this algorithm.

Radix sort is only applicable to integers not on strings.

If number is large then in count sort it will make a large array which means more space consuming.

### Dry Run:



## Quick Sort:

### Description:

It is also recursive algorithm. It works on the divide and conquers principle. It picked an element as a pivot and pivot can be last element, first element or any random element from array and after this do partition around the selected pivot. The array is reordered so that all values smaller than the pivot are moved before it and all values larger than the pivot are moved after it. This process is repeated for each subarray of smaller values as well as done separately for the subarray with greater values. This process continues until the entire array is sorted. Its time complexity is O(nlogn).

### Pseudo Code:

quickSort(A,p,r)

If (low < high)

pi = partition(A,p,r)

quickSort(A,p,pi-1)

quickSort(A,pi+1,r)

partition(A,p,r)

x = A[r]

i = p-1

for j = p to r-1

if A[j] <= x

i 🡨 i + 1

exchange A[i] with A[j]

exchange A[i+1] with A[r]

return **i + 1**

### Code in Python:

# Implementation of Quick Sort for both integers and strings

class QuickSort(sort):

    def \_\_init\_\_(self, arr):

        self.arr = arr

    def sortInt(self, reverse=False):

        return self.quickSort(self.arr,0,len(self.arr)-1,reverse)

    def quickSort(self,arr, low, high,reverse):

        if (low < high):

            pi = self.partition(self.arr, low, high)

            self.quickSort(self.arr, low, pi-1,reverse)

            self.quickSort(self.arr, pi+1, high,reverse)

        if(reverse == True):

            return self.arr[:: -1]

        else:

            return self.arr

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

        pivot = self.arr[high]

        i = (low - 1)

        for j in range(low,high):

            if(self.arr[j] < pivot):

                i += 1

                self.arr[i],self.arr[j] = self.arr[j],self.arr[i]

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

        return (i+1)

    def sortString(self, reverse=False):

        return self.quickSort(self.arr,0,len(self.arr)-1,reverse)

### Time Complexity Analysis:

quickSort(A,p,r)

If (low < high) (lg(n))

pi = partition(A,p,r) (n(lg(n)))

quickSort(A,p,pi-1) (lg(n))

quickSort(A,pi+1,r) (lg(n))

partition(A,p,r)

x = A[r] O(1)

i = p-1 O(1)

for j = p to r-1 (n)

if A[j] <= x (n-1)

i 🡨 i + 1 (n-1)

exchange A[i] with A[j] (n-1)

exchange A[i+1] with A[r] O(1)

return **i + 1**

### Three Strengths:

Quick Sort is applicable to both integers and strings.

Quick Sort is much faster than other algorithms in most of the cases.

Quick Sort doesn’t require additional memory.

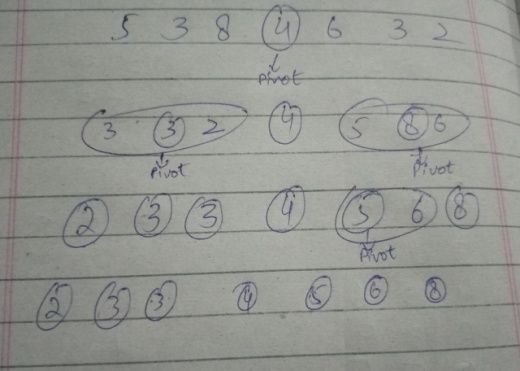
### Three Weaknesses:

Quick Sort is recursive algorithm even if it is the case where we it is difficult to apply recursion but if we are using quick sort we have to apply recursion.

If array is already sorted then its time complexity is n2 which is bad from many other sorting algorithms in same situation.

Quick Sort algorithm efficiency depends upon pivot if it chooses a bad pivot can also lead us worst time complexity (n2) and it is unstable

### Dry Run:



## Count Sort:

### Description:

It works by iterating through the input, counting the number of times each item occurs and storing that count of it in another array of length of maximum element of main array and on the same number index (Like if the number is 3 it will store its count on other on 3rd index). When we reach the end, we'll have the total counts for each number. Now that we know how many times each item appears, we can fill in our sorted array by looking at counts (Like if 1 appears one times it will place 1 in sorted array first and if 2 appears two times it will place 2 after 1 in sorted array 2 times and like this the whole array will be done and in the end we will have the array which is sorted). Its time complexity is O(n+k).

### Pseudo Code:

for i = 0 to arr.length()

count[arr[i]-Min] 🡨 count[arr[i]-Min] + 1

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

            count[i] = count[i] + count[i-1]

for i = arr.length() to 0:

            value 🡨 count[self.arr[i]-Min]

            value 🡨 value - 1

            count[self.arr[i]-Min] 🡨 count[self.arr[i]-Min] - 1

            arr2[value] 🡨 (self.arr[i]-Min) + Min

### Code in Python:

# Implementation of Count Sort for Integers

class CountSort(sort):

    def \_\_init\_\_(self, arr):

        self.arr = arr

    def sortInt(self, reverse=False):

        return self.countSort(self.arr,reverse)

    def countSort(self,arr,reverse):

        Max = max(arr)

        Min = min(arr)

        Max += 1

        count = [0] \* (Max-Min)

        for i in range(0,len(self.arr)):

            increment = 1

            count[self.arr[i]-Min] += increment

        for i in range(1,len(count)):

            count[i] = count[i] + count[i-1]

        arr2 = [0] \* len(self.arr)

        for i in reversed(range(0,len(self.arr))):

            value = count[self.arr[i]-Min]

            value -= 1

            count[self.arr[i]-Min] -= 1

            arr2[value] = (self.arr[i]-Min) + Min

        if(reverse == True):

            return arr2[:: -1]

        else:

            return arr2

### Time Complexity Analysis:

for i = 0 to arr.length() (n)

count[arr[i]-Min] 🡨 count[arr[i]-Min] + 1 (n-1)

for i in range(1,len(count)): (n)

            count[i] = count[i] + count[i-1] (n-1)

for i = arr.length() to 0: (n)

            value 🡨 count[self.arr[i]-Min] (n-1)

            value 🡨 value - 1 (n-1)

            count[self.arr[i]-Min] 🡨 count[self.arr[i]-Min] -1 (n-1)

            arr2[value] 🡨 (self.arr[i]-Min) + Min (n-1)

### Three Strengths:

Count Sort is helpful where we want to sort array with linear time complexity O(n).

Count Sort has linear time complexity same as in radix sort.

Count Sort elements by counting them in this way we got count of any specific number that is present in an array.

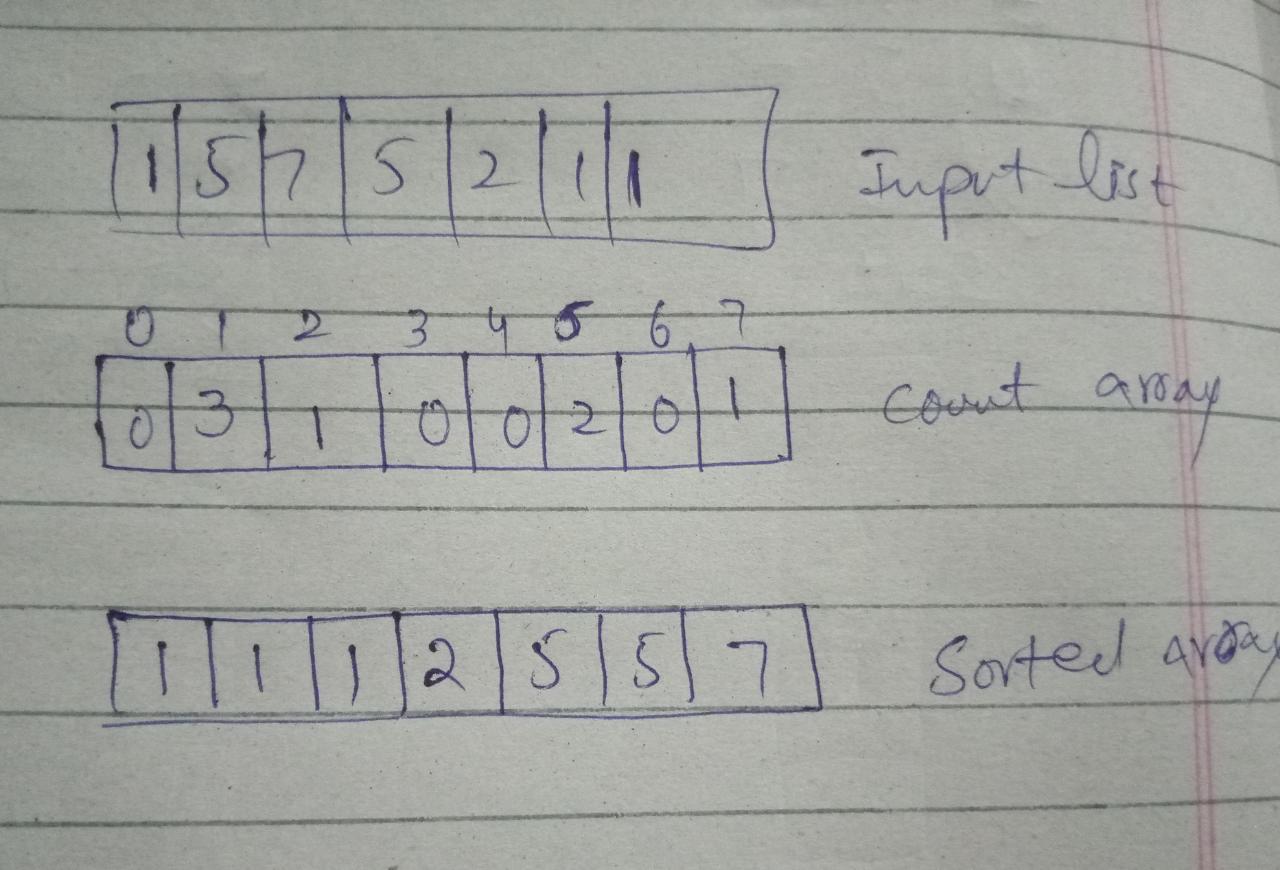
### Three Weaknesses:

Count Sort is applicable to only integers not on strings.

If the integers are negatively large then it will make a large array which means more space consuming.

Time complexity and space complexity become worst if the input array elements is large.

### Dry Run:



## Pancake Sort:

### Description:

It first finds maximum from the given array and swap it on the last index of array and decrement array length each time. In this way it starts building a sorted array at the end. On every iteration, we find maximum and swap it on the last index of array until the whole array is sorted. Its time complexity is best case is O(n) and in worst case O(n2 ).

### Pseudo Code:

while(arr.length() != 0):

       for i=0 to arr.length():

             if(arr[i] > Maximum):

                   Maximum 🡨 arr[i]

                   index 🡨 i

       arr[arr.length() - 1] , arr[index] = arr[index], arr[arr.length() - 1]

       arr.length() 🡨 arr.length - 1

### Code in Python:

# Implementation of Pancake Sort for both integers and strings

class pancakeSort(sort):

    def \_\_init\_\_(self, arr):

        self.arr = arr

    def sortInt(self, reverse=False):

        return self.PancakeSortInt(self.arr, reverse)

    def PancakeSortInt(self, arr, reverse):

        length = len(arr)

        while(length != 0):

            Maximum = 0

            for i in range(0,length):

                if(self.arr[i] > Maximum):

                    Maximum = self.arr[i]

                    index = i

            self.arr[length - 1] , self.arr[index] = self.arr[index], self.arr[length - 1]

            length -= 1

        if(reverse == True):

            return self.arr[:: -1]

        else:

            return self.arr

    def sortString(self, reverse=False):

        return self.PancakeSort(self.arr, reverse)

    def PancakeSort(self, arr, reverse):

        length = len(arr)

        while(length != 0):

            Maximum = 0

            for i in range(0,length):

                if(str(self.arr[i]) > str(Maximum)):

                    Maximum = self.arr[i]

                    index = i

            self.arr[length - 1] , self.arr[index] = self.arr[index], self.arr[length - 1]

            length -= 1

        if(reverse == True):

            return self.arr[:: -1]

        else:

            return self.arr

### Time Complexity Analysis:

while(arr.length() != 0): (n)

       for i=0 to arr.length(): n(n-1)

             if(arr[i] > Maximum): (n-2)

                   Maximum 🡨 arr[i] (n-2)

                   index 🡨 i (n-2)

       arr[arr.length() - 1] , arr[index] = arr[index], arr[arr.length() - 1] (n)

       arr.length() 🡨 arr.length – 1 (n)

### Three Strengths:

Pancake finds maximum in array and place at last of index this is the advantage of pancake instead of swapping.

Pancake doesn’t compare values like other comparison based algorithms do.

Pancake algorithm also minimizes number of operations.

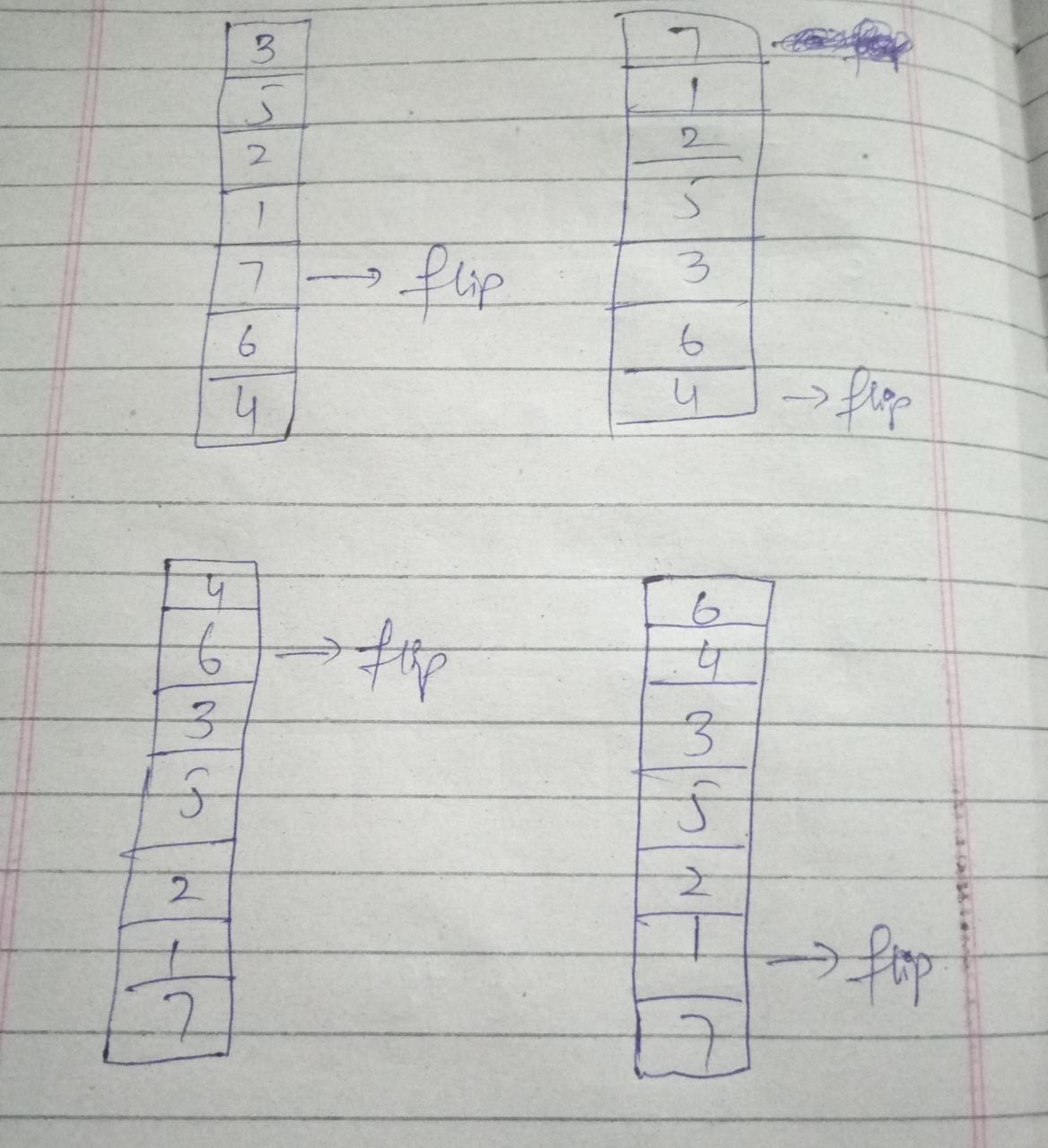
### Three Weaknesses:

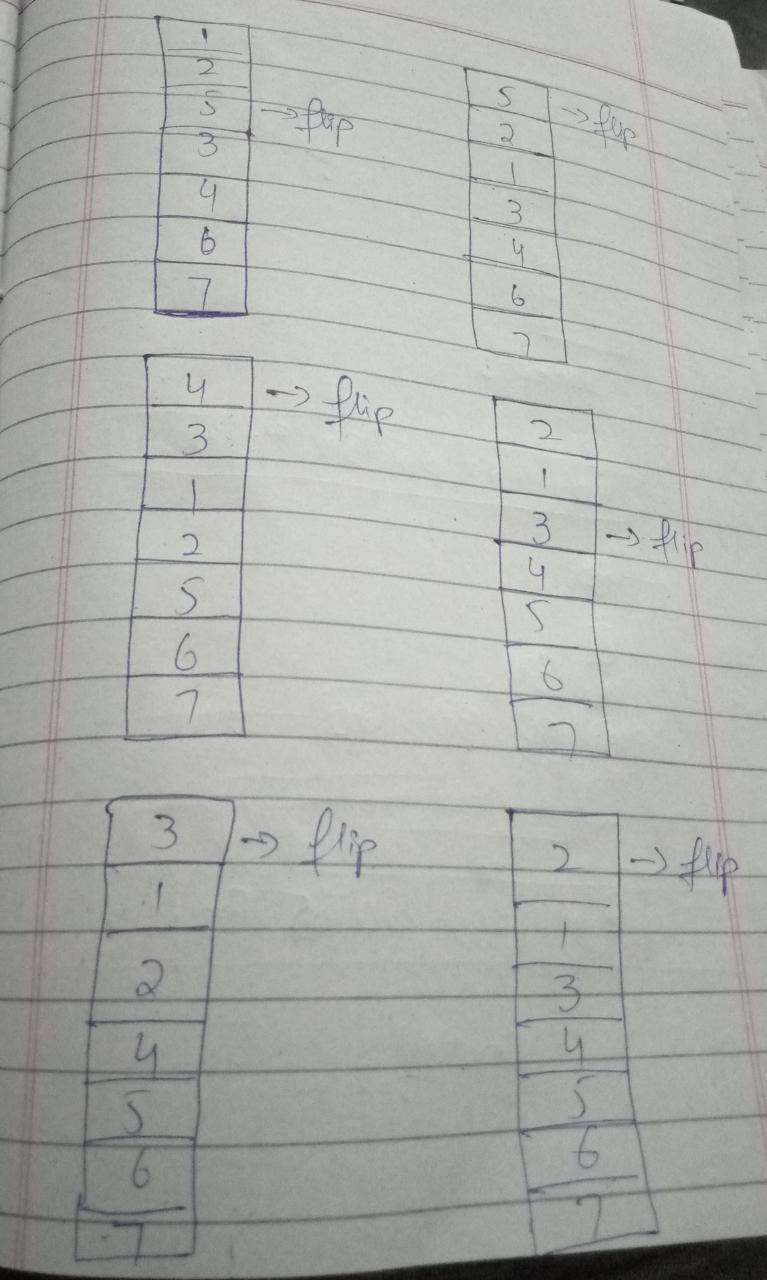
Pancake sort is not a stable sort algorithm.

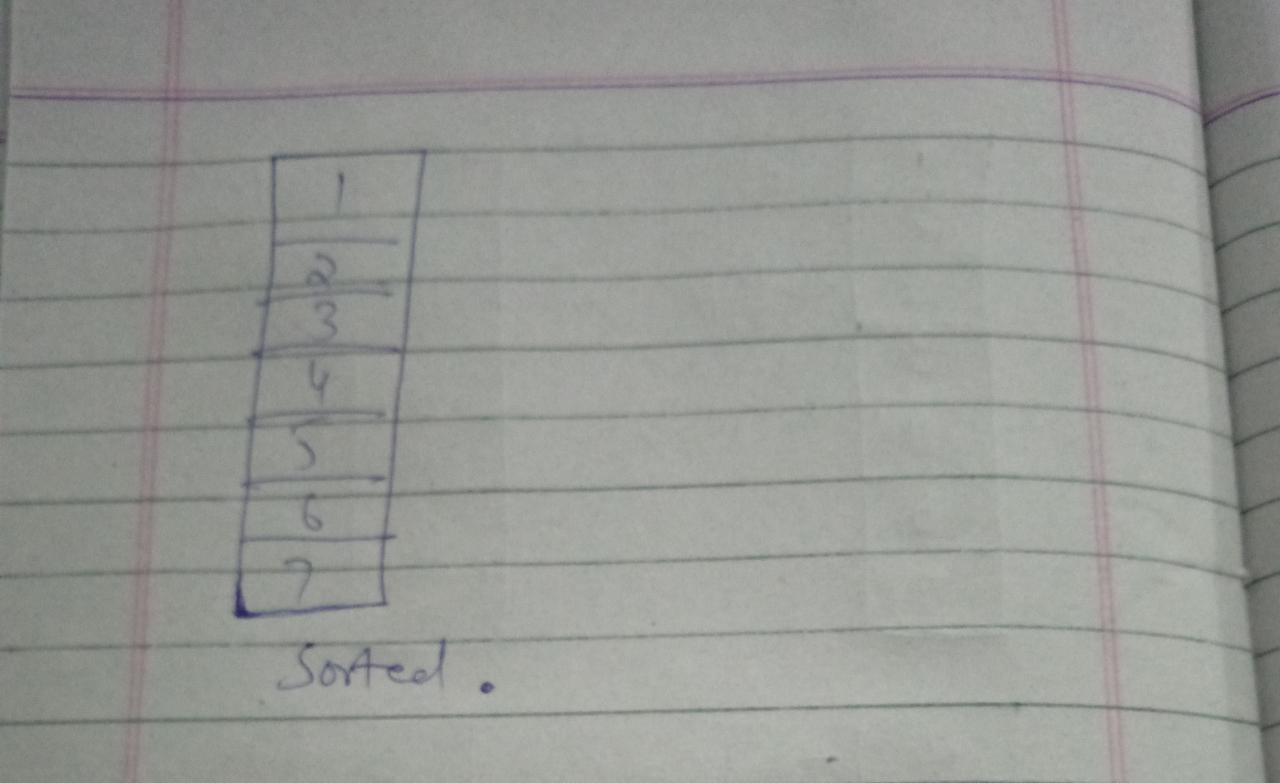
Its time complexity is (n2) in most of the cases.

It kept on swap the number randomly in array.

### Dry Run:







## Patience Sort:

### Description:

It takes first element of array and compare it with next element if it is smaller it put it with that number in new list next to it and if number is larger it forms a new list and put number in that now it will check for both list if the number is smaller than it will always prefer first list and put on next index in it now it will take minimum value from every list and compare it and place smallest element first in list and this process continues until whole array is being sort.

### Pseudo Code:

for i=0 to arr.length():

       for j=0 to list1:

             if j[len(j)-1] > arr[i]):

                   j 🡨 append(arr[i])

for i=0 to arr.length():

            index 🡨 0

            c 🡨 len(list1[0])

            for i=0 to list1.length():

                if(len(list1[i]) > c):

                    index 🡨 i

            compare1 🡨 list1[index][len(list1[index]) - 1]

            for j=0 to list1.length():

                if(list1[j] != []):

                    compare2 🡨 list1[j][len(list1[j])-1]

                    if(compare2 < compare1):

                        compare1 🡨 compare2

                        index 🡨 j

### Code in Python:

# Implementation of Patience Sort for both integers and strings

def sortInt(self, reverse=False):

        return self.PatienceSortInt(self.arr, reverse)

    def PatienceSortInt(self,arr,reverse):

        list1 = []

        list1.append([arr[0]])

        for i in range(1,len(arr)):

            appended = False

            for j in list1:

                if j[len(j)-1] > arr[i]:

                    j.append(arr[i])

                    appended = True

                    break

            if (appended == False):

                list1.append([arr[i]])

        newArray = []

        for i in range(0,len(arr)):

            index = 0

            c = len(list1[0])

            for i in range(len(list1)):

                if(len(list1[i]) > c):

                    index = i

            compare1 = list1[index][len(list1[index]) - 1]

            for j in range(0,len(list1)):

                if(list1[j] != []):

                    compare2 = list1[j][len(list1[j])-1]

                    if(compare2 < compare1):

                        compare1 = compare2

                        index = j

            if(str(compare1) > str(0)):

                newArray.append(compare1)

                list1[index].pop(len(list1[index])-1)

            else:

                newArray.append(compare1)

                list1[index].pop(len(list1[index])-1)

        if(reverse == True):

            return newArray[:: -1]

        else:

            return newArray

    def sortString(self, reverse=False):

        return self.PatienceSortInt(self.arr, reverse)

### Time Complexity Analysis:

for i=0 to arr.length(): (n)

       for j=0 to list1: (n(n-1))

             if j[len(j)-1] > arr[i]): (n-2)

                   j 🡨 append(arr[i]) (n-2)

for i=0 to arr.length(): (n)

            index 🡨 0 (n-1)

            c 🡨 len(list1[0]) (n-1)

            for i=0 to list1.length(): (n(n-1))

                if(len(list1[i]) > c): (n-2)

                    index 🡨 i (n-2)

            compare1 🡨 list1[index][len(list1[index]) - 1] (n-1)

            for j=0 to list1.length(): (n(n-1))

                if(list1[j] != []): (n-2)

                    compare2 🡨 list1[j][len(list1[j])-1] (n-3)

                    if(compare2 < compare1): (n-2)

                        compare1 🡨 compare2 (n-3)

                        index 🡨 j (n-3)

### Three Strengths:

Patience Sort is applicable on both integers and strings.

Patience Sort allows few numbers of piles/ buckets as much as possible.

Patience Sort put numbers in buckets in such way that they are already sorted.

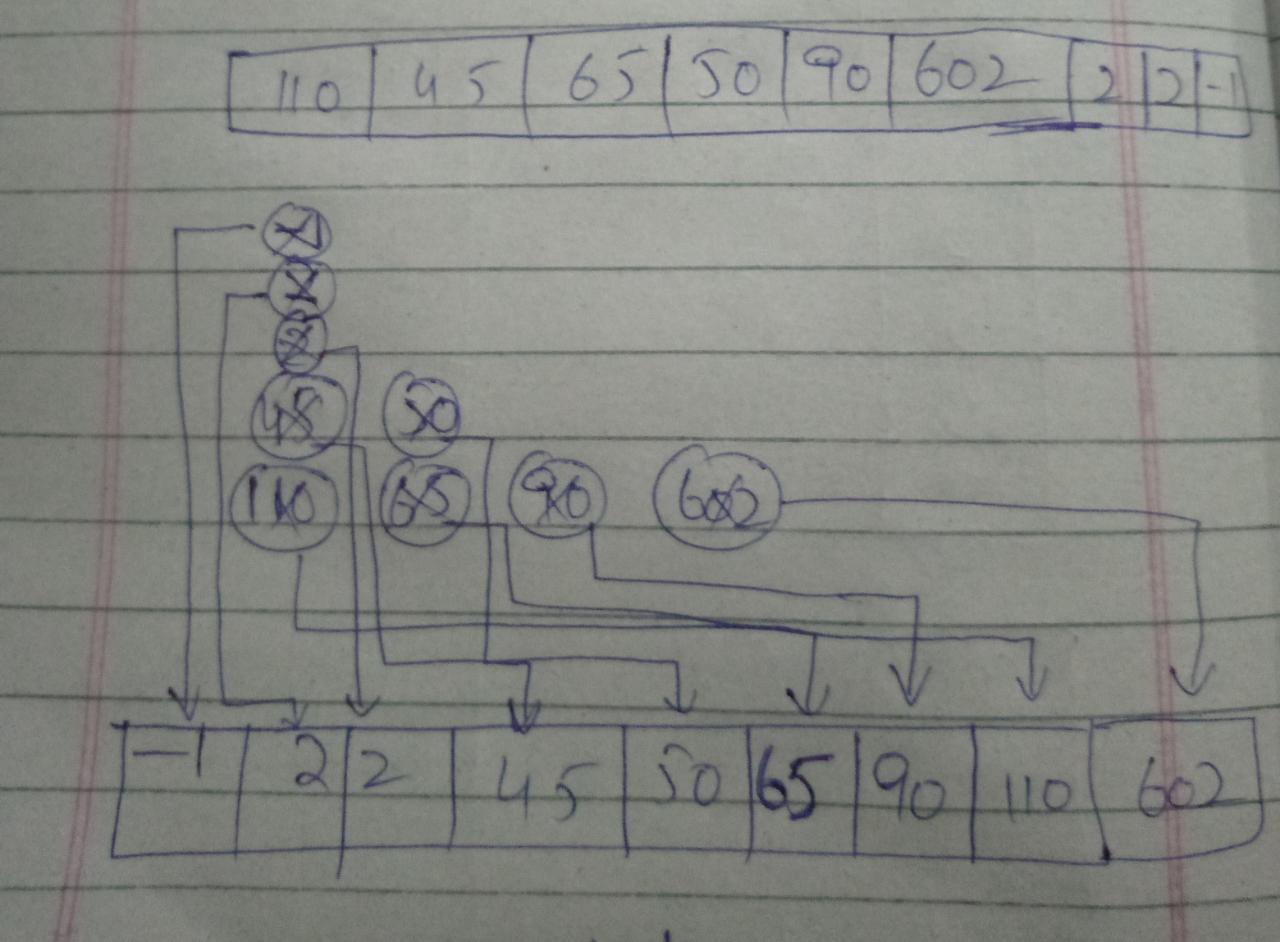
### Three Weaknesses:

Patience Sort allows few numbers of piles if we are given a large array can contradict this statement.

Patience Sort is slower for the larger number of elements array.

Patience Sort is worst in most of the cases.

### Dry Run:



# Multi-level Sorting:

In this algorithm, when sorting is applied, first data is sorted simply and the column w.r.t. which data is sorted, the column header is saved in a text file. Again when sorting is performed, the previous sorted data and column header is loaded in the file and the previous sorted data is divided into subarrays containing same values. Then on each array sorting is performed. And a 2 level sorting is done. When again sorting is performed the steps are repeated again. In this way, multi-level sorting is done.

## Code in Python:

class MultiLevelSorting(Sort):

    def \_\_init\_\_(self, arr, data) -> None:

        super().\_\_init\_\_(arr)

        self.data = data

    def sort(self, reverse=False):

        sortedData = []

        if self.data == "":

            from sorting.mergeSort import MergeSort

            merge = MergeSort(self.arr)

            sortedData = merge.sort(reverse)

        else:

            subarray = []

            if type(self.arr[0][self.data]) == str:

                item = str(self.arr[0][self.data])[0]

            else:

                item = self.arr[0][self.data]

            for i in range(self.length - 2):

                if item ==  str(self.arr[i + 1][self.data])[0] and type(self.arr[0][self.data]) == str:

                    subarray.append(self.arr[i])

                elif item == self.arr[i + 1][self.data] and type(self.arr[i + 1][self.data]) == int:

                    subarray.append(self.arr[i])

                else:

                    from sorting.mergeSort import MergeSort

                    merge = MergeSort(subarray)

                    sorted = merge.sort(reverse)

                    for line in sorted:

                        sortedData.append(line)

                    subarray = []

                    if type(self.arr[0][self.data]) == str:

                        item = str(self.arr[i][self.data])[0]

                    else:

                        item = self.arr[i][self.data]

        df = pd.DataFrame(sortedData)

        df.to\_csv('CSV\_Files/multiLevelSortingOutput.csv', index=False, encoding='utf-8', header=True)

        f = open(r"TextFiles\multiSortingData.txt", "w")

        f.writelines([self.Key+"\n"])

        f.close()

        return sortedData

# Searching Algorithms:

## Linear Search:

### Description:

In this whole algorithm the value is to be searched in the whole array and where the value will be found it will return it with a time complexity of O(n).

### Pseudo Code:

for i = 0 to A.Length  
 if value == A[i]  
 return i

### Code in Python:

class LinearSearch(Search):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    def search(self, value):

        result = []

        for data in self.arr:

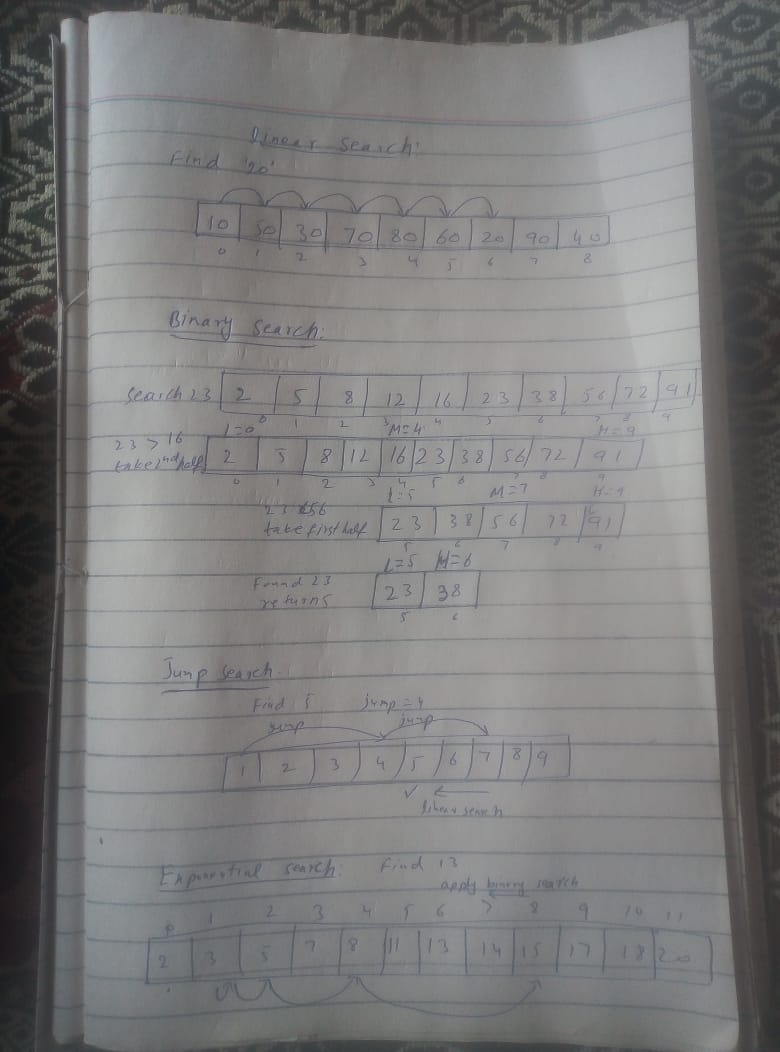
            if (value == str(data[Search.Key])):

                result.append(data)

### Time Complexity Analysis:

for i = 0 to A.Length n - 1  
 if value == A[i] n – 2  
 return i

### Dry run:



## Binary Search:

### Description:

Binary Search is a recursive algorithm applicable only on sorted arrays. Also can be implemented in iterative method. As the array is sorted, it is divided into two arrays by a mid-value and then the value which is to be searched is compared with the mid value of the array. If the value matched, then it will return the index otherwise if value which to be searched is greater than the mid value of the array then the method will be applied again recursively on the right sub array and vice versa. The method is repeated until the value is found or no value found.

### Pseudo Code:

I wrote the iterative algorithm.

Binary\_Search(A, value, left, right)  
 data 🡨 []  
 while( left <= right)  
 mid = left + (right – left) // 2  
 if value == A[mid]  
 return mid  
 else if value < A[mid]  
 right = mid – 1  
 else  
 left = mid + 1

### Code in Python:

class BinarySearch(Search):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    def search(self, value):

        if type(self.arr[0][self.Key]) == int:

            try:

                value = int(value)

            except:

                pass

        else:

            value = str(value)

        self.bsearch(value, 0, self.length - 1)

    def bsearch(self, value, left, right):

        data = []

        try:

            while left <= right:

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

                if value == self.arr[mid][self.Key]:

                    data.append(self.arr[mid])

                    break

                elif value < self.arr[mid][self.Key]:

                    right = mid - 1

                else:

                    left = mid + 1

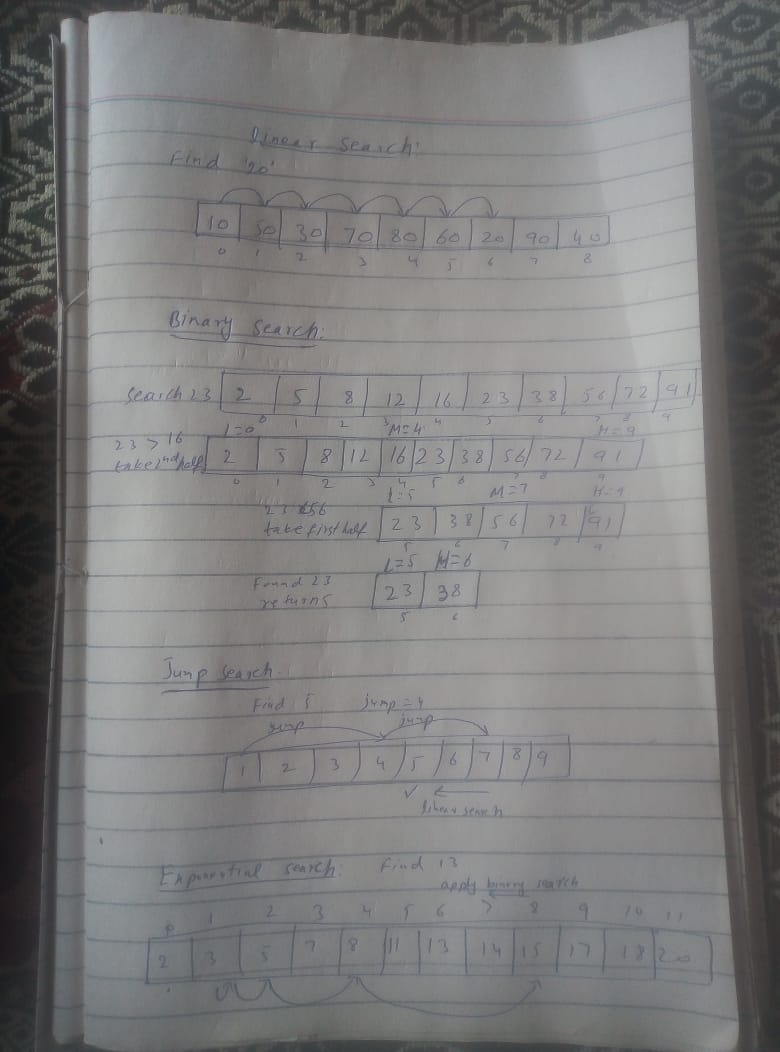
        except:

            pass

### Time Complexity Analysis:

Binary\_Search(A, value, left, right)  
 data 🡨 [] c  
 while( left <= right) lgn  
 mid = left + (right – left) // 2 lg(n – 1)  
 if value == A[mid] lg(n – 1)  
 return mid lg(n – 1)  
 else if value < A[mid] lg(n – 1)  
 right = mid – 1 lg(n – 1)  
 else lg(n – 1)  
 left = mid + 1 lg(n – 1)

### Dry run:



## Jump Search:

### Description:

Jump search also works on sorted array in which an interval is taken of any value. Then the value which is to be searched is compared with the array in intervals and the interval in which the value exists is found and then linear search is applied on the that interval to find the exact index.

### Pseudo Code:

Interval 🡨 5  
result 🡨 []  
while(interval < A.length and A[interval] <= value)  
 Interval += 5  
high = min 🡨 A.length , Interval  
for i = Interval – 5 to high  
 if value == A[i]  
 return i

### Code in Python:

class JumpSearch(Search):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    def search(self, value):

        high = 3

        result = []

        while(high < self.length and self.arr[high][Search.Key] <= value):

            high += 3

        h = min(self.length,high)

        for i in range(high - 3, h):

            if (str(value) == str(self.arr[i][Search.Key])):

                result.append(self.arr[i])

        df = pd.DataFrame(result)

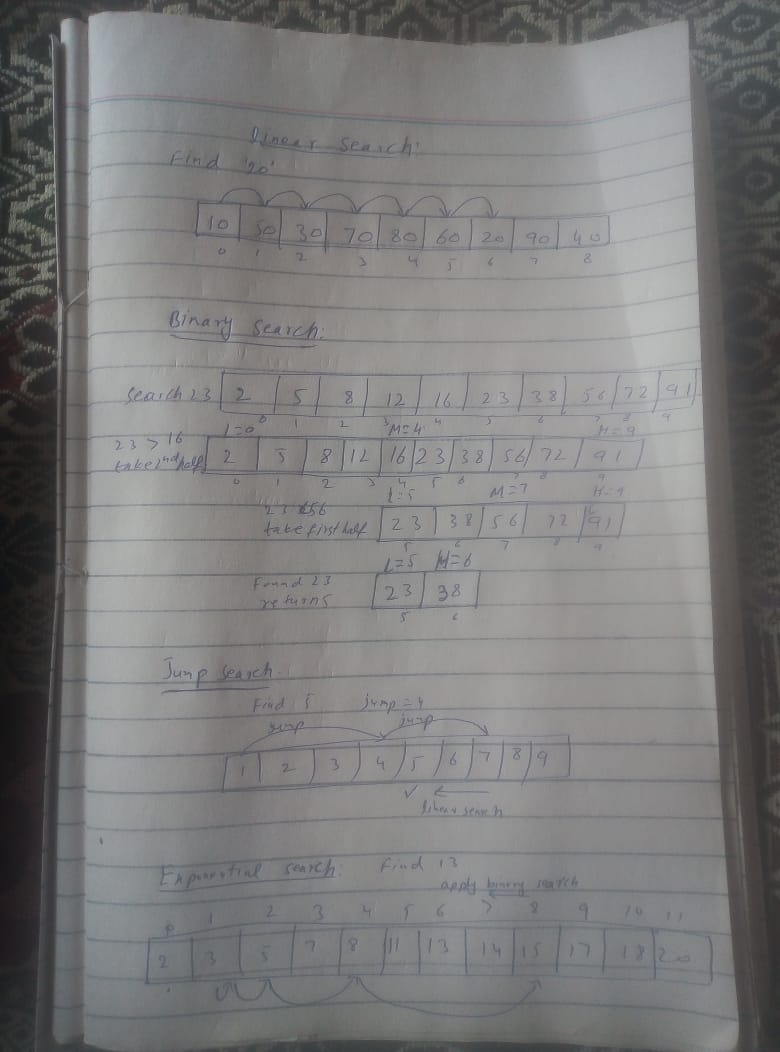
        df.to\_csv('CSV\_Files/output.csv', index=False,

                  encoding='utf-8', header=True)

### Time Complexity Analysis:

Interval 🡨 5  
result 🡨 []  
while(interval < A.length and A[interval] <= value) n  
 Interval += 5 n - 1  
high = min 🡨 A.length , Interval c  
for i = Interval – 5 to high n  
 if value == A[i] n - 1  
 return i

### Dry Run:



## Exponential Search:

### Description:

This algorithm also works on the sorted array in which exponential interval of 2 is taken and the interval is to be found in which the value, that is to be searched, exists. Then, binary search is applied on that interval.

### Pseudo Code:

n = 1  
while (n < A.length and A[n] <= value  
 n \*= 2  
Binary\_Search(value, n / 2, min 🡨 n, A.length – 1)

### Code in Python:

class ExponentialSearch(Search):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    def search(self, value):

        if type(self.arr[0][self.Key]) == int:

            try:

                value = int(value)

            except:

                pass

        else:

            value = str(value)

        n = 1

        while (n < self.length and self.arr[n][self.Key] <= value):

            n \*= 2

        print(n)

        from searching.binarySearch import BinarySearch

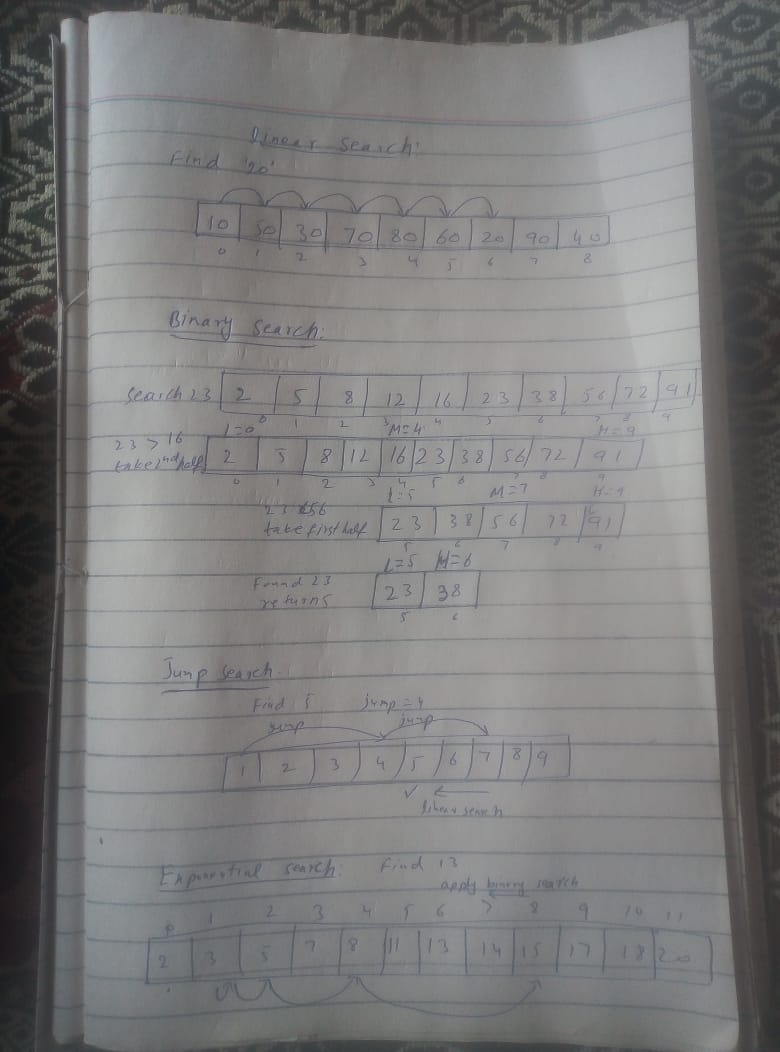
        b\_search = BinarySearch(self.arr)

        b\_search.bsearch(value, n // 2, min(n, self.length - 1))

### Time Complexity Analysis:

while (n < A.length and A[n] <= value n  
 n \*= 2 n - 1  
Binary\_Search(value, n / 2, min 🡨 n, A.length – 1) c

### Dry Run:



# Composite Searching:

This algorithm applies on the input values containing “and”, “or” and “not”. The value should contain only one keyword like “a and b”, “a or b” or “not a”.

## Code in Python:

class CompositeFilter(Search):

    def \_\_init\_\_(self, arr) -> None:

        super().\_\_init\_\_(arr)

    def search(self, value):

        lgOp = ""

        idx = 0

        value = value.split(" ")

        if "and" in value:

            idx = value.index("and")

            lgOp = "and"

        elif "or" in value:

            idx = value.index("or")

            lgOp = "or"

        elif "not" in value and value.index("not") == 0:

            lgOp = "not"

        q1, q2 = "", ""

        for i in range(idx):

            q1 += value[i] + " "

        for i in range(idx + 1, len(value)):

            q2 += value[i] + " "

        q1, q2 = q1.strip(), q2.strip()

        print(q1, q2)

        data = []

        for j in self.arr:

            text = str(j["Name"])+" "+j["Id"]+" "+str(j["Price"])+" "+str(j["Pages"])+" "+j["Author"]+" "+j["Language"]+" "+j["Type"]

            if lgOp == "and":

                if ((q1 in text) and (q2 in text)) == True:

                    data.append(j)

            elif lgOp == "or":

                if ((q1 in text) or (q2 in text)) == True:

                    data.append(j)

            elif lgOp == "not":

                if q2 not in text:

                    data.append(j)

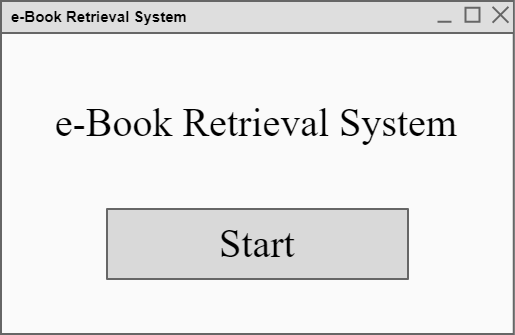
        df = pd.DataFrame(data)

        df.to\_csv('CSV\_Files/output.csv', index=False, encoding='utf-8', header=True)

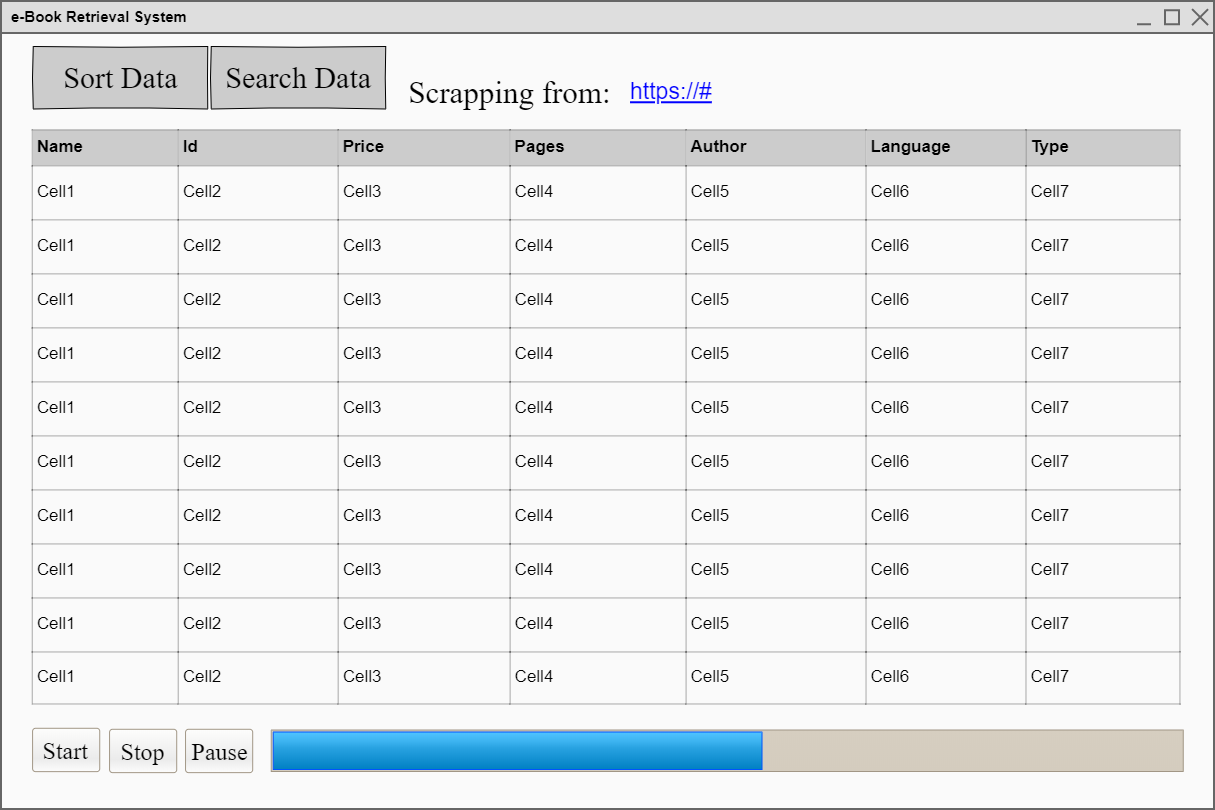
# UI:

## Wireframes:

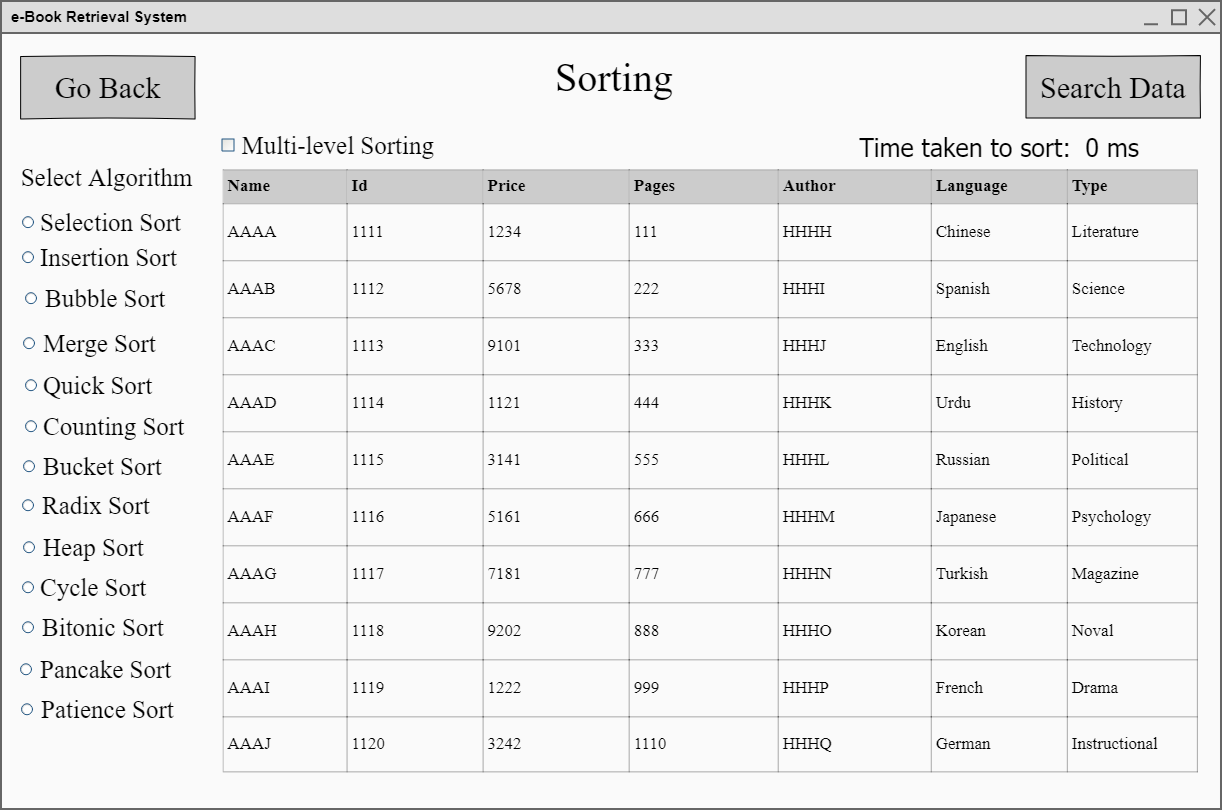
### Start Window:



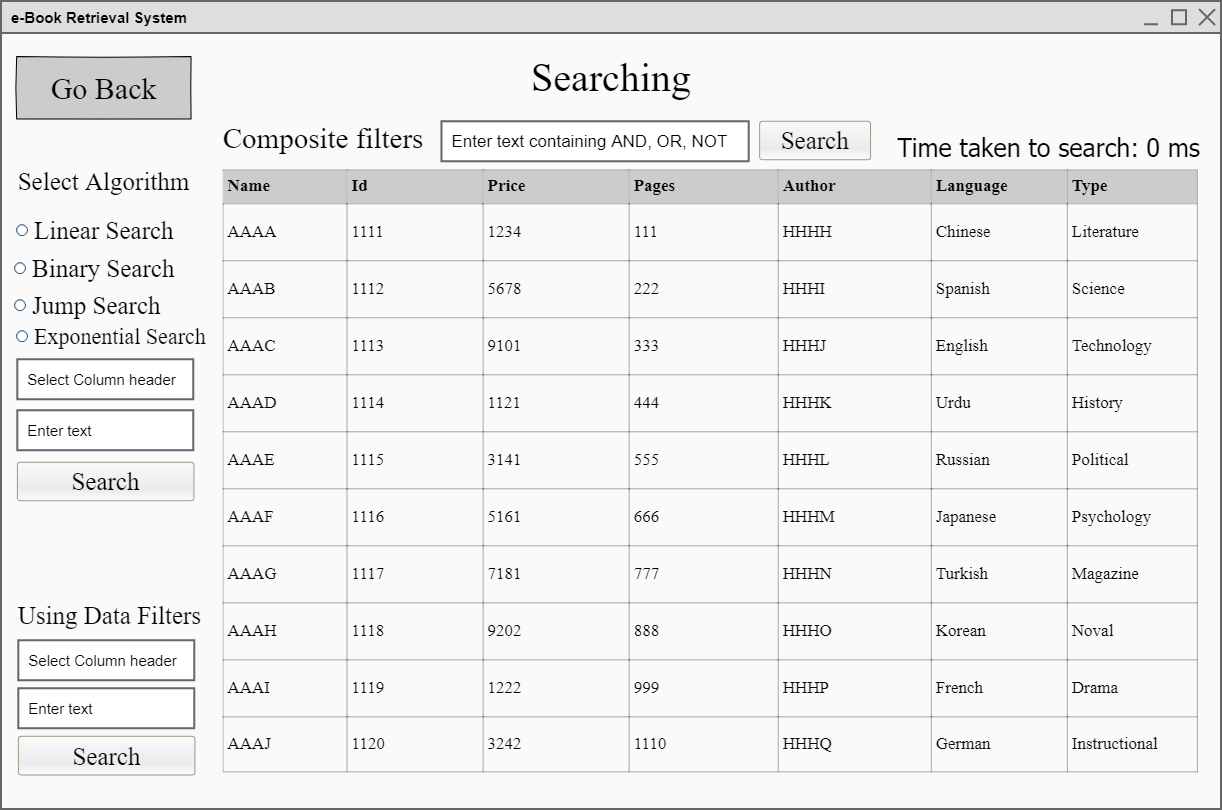
### Scrap Window:



### Sort Window:

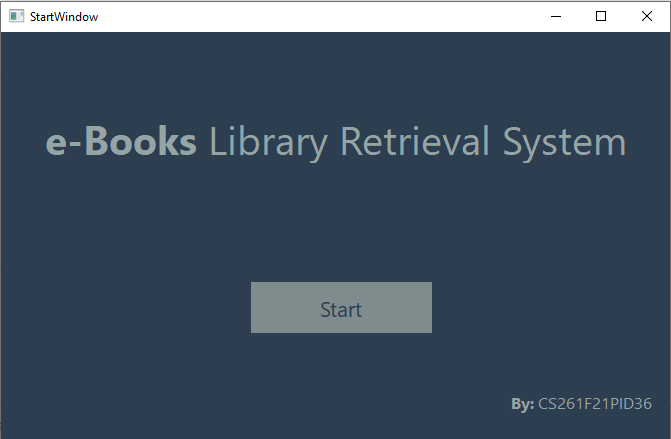


### Search Window:



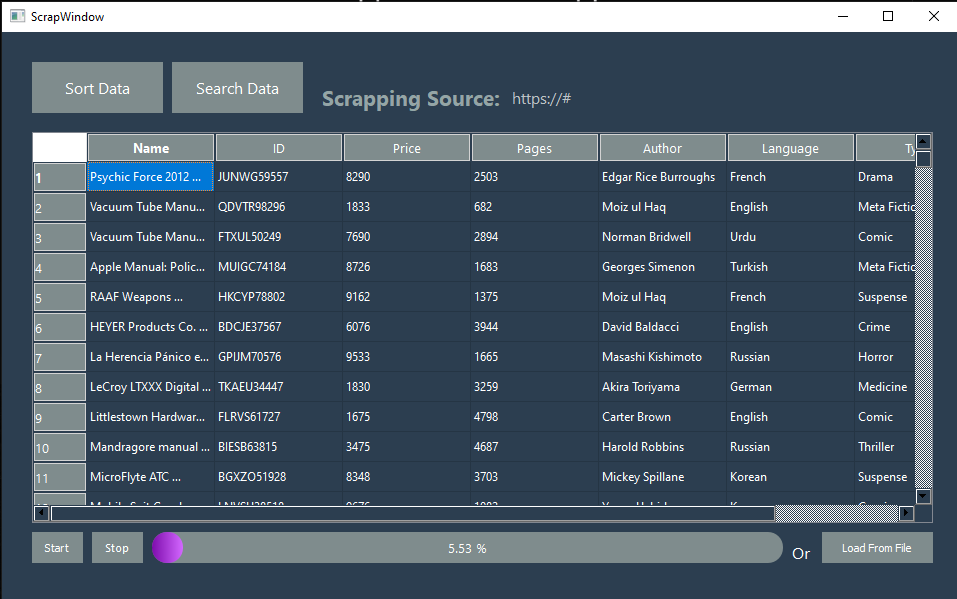
## Implementation:

### Start Window:



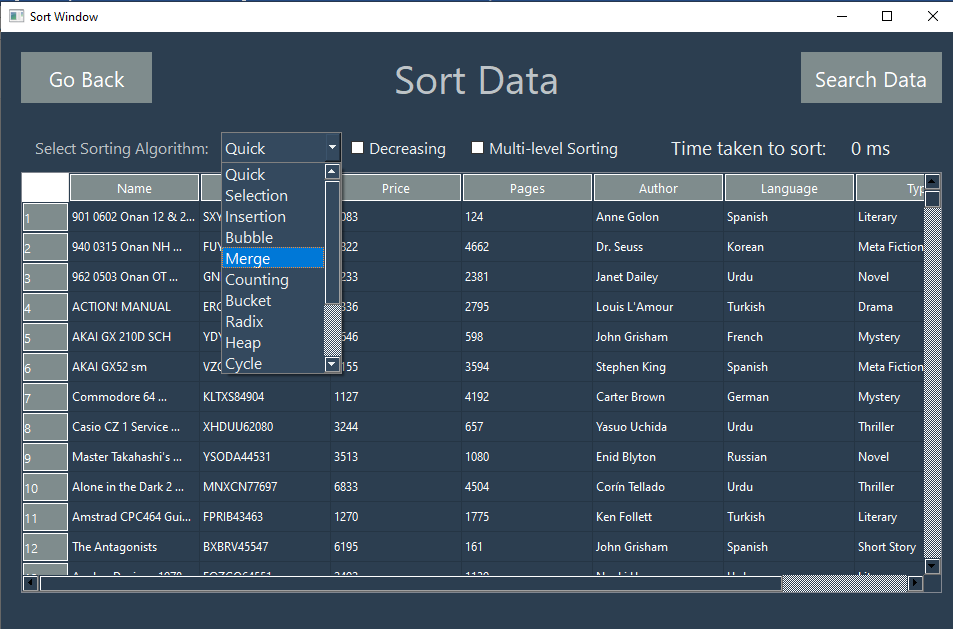
|  |  |  |
| --- | --- | --- |
| Component Name | Type | Description |
| label | QLabel | Label showing the title of project (e.Books Library Retrieval System) |
| startBtn | QPushButton | When clicked on that button new scrap window is opened. |
| label | QLabel | Label showing the project ID |

### Scrap Window:



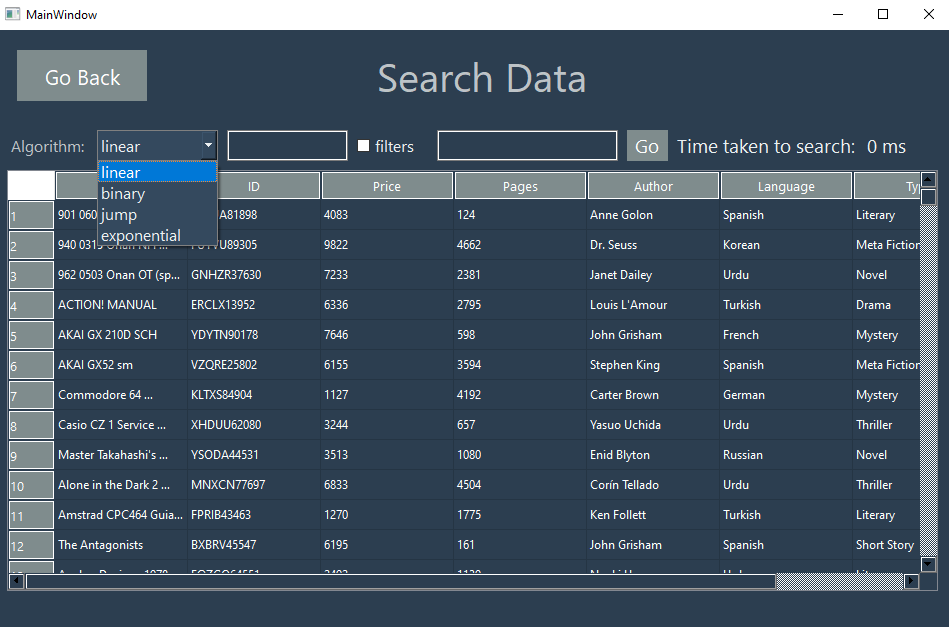
|  |  |  |
| --- | --- | --- |
| Component Name | Type | Description |
| sortDataBtn\_ScrapeWindow | QPushButton | Button to go to the Sort Window |
| searchDataBtn\_ScrapeWindow | QPushButton | Button to go to the Search Window |
| label\_2 | QLabel | label to show the scrapping source |
| ScrapeWindowTable | QTableWidget | Table to show the scrapped data |
| startBtn\_ScrapeWindow | QPushButton | Button to start the scrapping |
| stopBtn\_ScrapeWindow | QPushButton | Button to stop the scrapping |
| progressBar | QProgressBar | Progress Bar to show the % of scrapped data |
| loadBtn\_ScrapeWindow | QProgressBar | Button to load data from file |

### Sort Window:



|  |  |  |
| --- | --- | --- |
| Component Name | Type | Description |
| goBack\_btn | QProgressBar | Button to go back to the scrapping window |
| searchData\_btn | QProgressBar | Button to go to the search window |
| algo\_comboBox | QComboBox | Combo box showing all the sorting algorithms |
| decre\_incre\_cb | QCheckBox | To sort data in ascending or decreasing order |
| multi\_level\_sorting\_cb | QCheckBox | To implement multi level sorting |
| tableWidget | QTabelWidget | To show data on table |
| show\_time\_label | QLabel | To show time that is taken to sort data |

### Search Window:

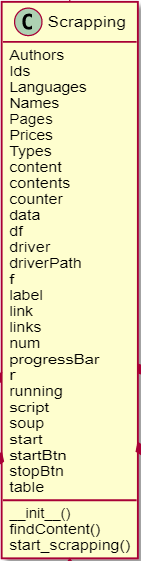


|  |  |  |
| --- | --- | --- |
| Component Name | Type | Description |
| go\_back\_btn | QPushButton | To back to the sort window |
| algo\_combo\_box | QComboBox | To select search algorithm |
| search\_input | QLineEdit | To take value which to be searched |
| filters\_checkBox | QCheckBox | To filter the data by value taken |
| search\_input\_composite | QLineEdit | To take value which to be searched containing and, or and not |
| search\_input\_composite\_btn | QPushButton | To search value(composite input) from the whole data |
| show\_time\_label | QLabel | To show time taken to search value from the data |
| tableWidget | QTableWidget | To show data on table |

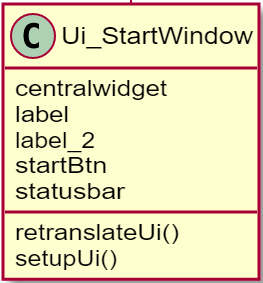
# Integration:

## Class Diagrams:

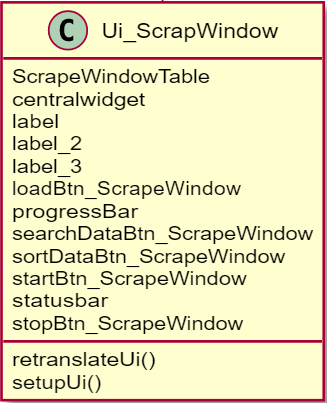
### Scrapping.py



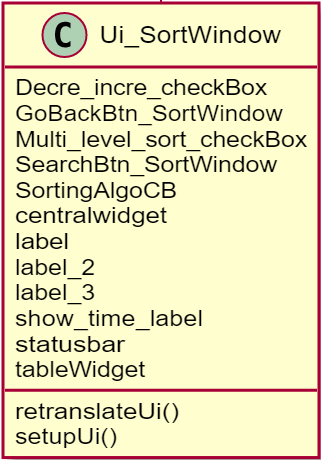
### Start\_Window\_UI.py



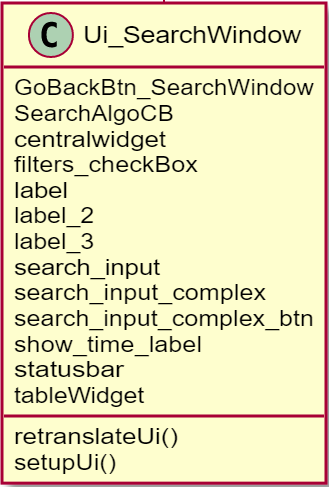
### Scrap\_Window\_UI.py



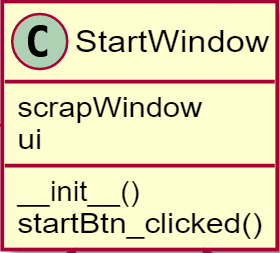
### Sort\_Window\_UI.py



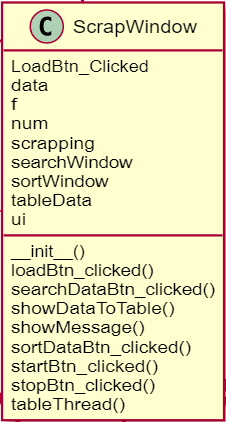
### Search\_Window\_UI.py



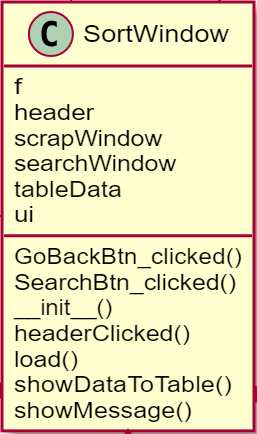
### Start\_Window\_UI\_Controller.py



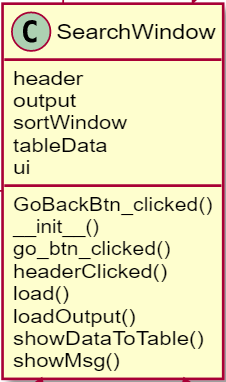
### Scarp\_Window\_UI\_Controller.py



### Sort\_Window\_UI\_Controller.py

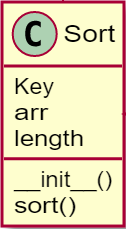


### Search\_Window\_UI\_Controller.py

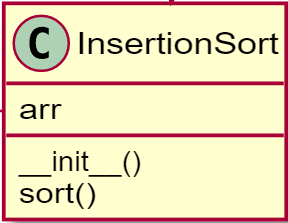


### Sort.py

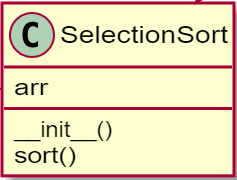
This is used as an abstract class for all other sorting algorithms.



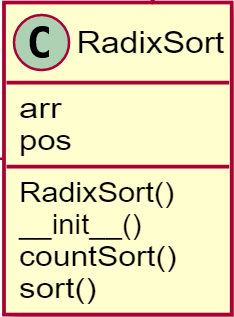
### Insertion\_Sort.py



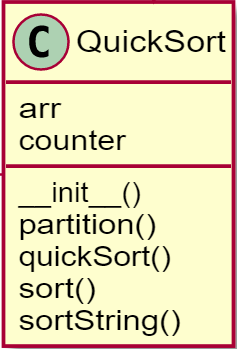
### Selection\_Sort.py



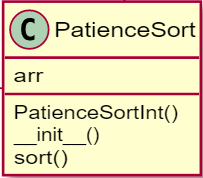
### Radix\_Sort.py



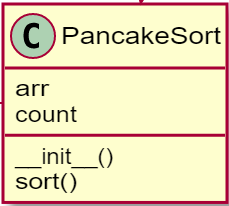
### Quick\_Sort.py



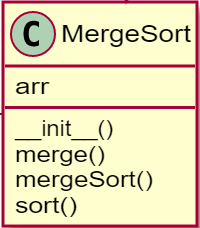
### Patience\_Sort.py



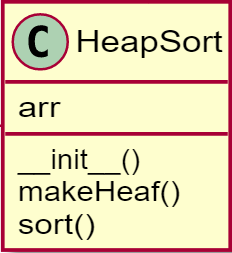
### Pancake\_Sort.py



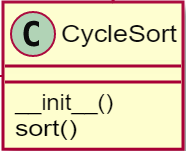
### Merge\_Sort.py



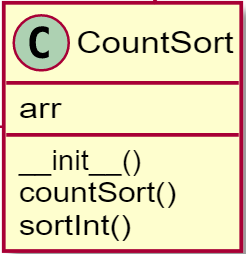
### Heap\_Sort.py



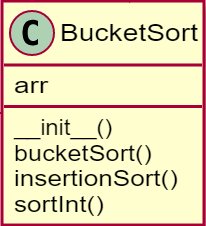
### Cycle\_Sort.py



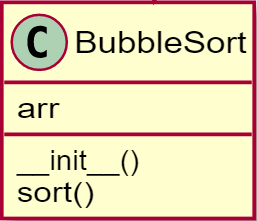
### Count\_Sort.py



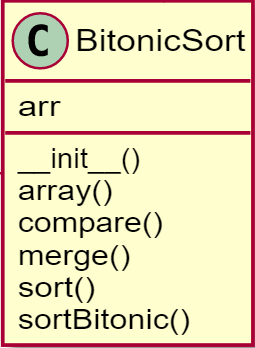
### Bucket\_Sort.py



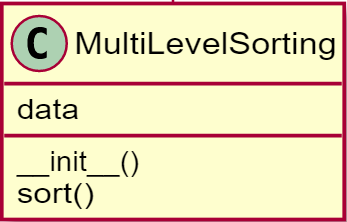
### Bubble\_Sort.py



### Bitonic\_Sort.py

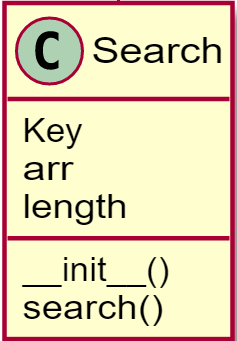


### Multi-level\_Sorting.py

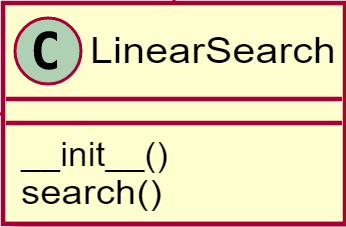


### Search.py

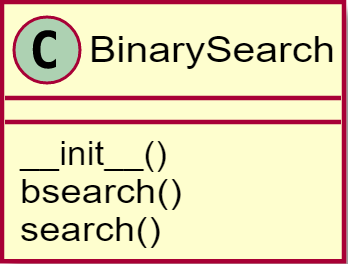
This class also used as an abstract class for all searching algorithms.



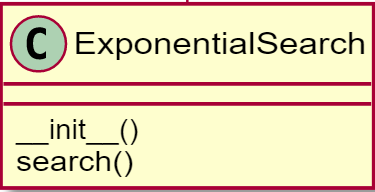
### Linear\_Search.py



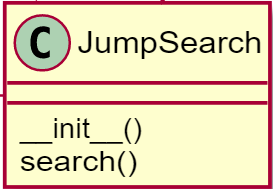
### Binary\_Search.py



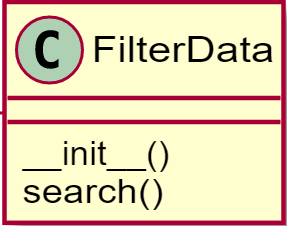
### Exponential\_Search.py



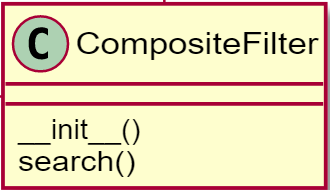
### Jump\_Search.py



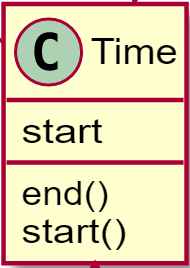
### Filter\_Data.py



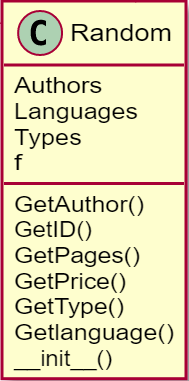
### Composite\_Filter.py



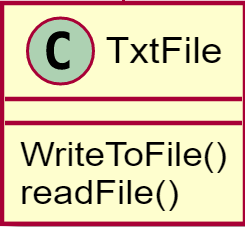
### Time.py



### Random.py



### TxtFile\_Handling.py



# Collaboration:

We used github desktop app to collaborate with each other while creating project. We divided our project in such a way that each group member done equal work. History of commits is shown in the figure below.

