Mobile and Computer Accessories Scrapper

Midterm Project Requirement Document



Project Supervisor

Mr. Samyan Qayyum Wahla

Project Members (GR-51)

|  |  |
| --- | --- |
| Syed Ushher Ahmad | 2020-CS-107 |
| Muhammad Taha | 2020-CS-126 |

Department of Computer Science

University of Engineering and Technology, Lahore

Pakistan

# Table of Contents

[Table of Contents 1](#_Toc88211228)

[List of Figures 2](#_Toc88211229)

[Project Description: 3](#_Toc88211230)

[Motivation and Impact: 4](#_Toc88211231)

[Project Features: 4](#_Toc88211232)

[Scrapper Functionalities: 5](#_Toc88211233)

[Functionality: 5](#_Toc88211234)

[User Interface: 6](#_Toc88211235)

[Collaboration: 6](#_Toc88211236)

[Work done by 2020-CS-107: 6](#_Toc88211237)

[Work done by 2020-CS-126: 6](#_Toc88211238)

[Graphical User Interface: 7](#_Toc88211239)

[Use Case # 1 7](#_Toc88211240)

[*Use Case # 2(Search)* 7](#_Toc88211241)

[Use Case # 3 8](#_Toc88211242)

[Use Case # 4 9](#_Toc88211243)

[Sorting Techniques: 10](#_Toc88211244)

[Counting Sort: 10](#_Toc88211245)

[Radix Sort: 14](#_Toc88211246)

[Bucket Sort: 18](#_Toc88211247)

[Heap Sort: 21](#_Toc88211248)

[Tim Sort: 25](#_Toc88211249)

[Insertion Sort: 29](#_Toc88211250)

[Bubble Sort: 32](#_Toc88211251)

[Selection Sort: 34](#_Toc88211252)

[Merge Sort: 37](#_Toc88211253)

[Quick Sort: 40](#_Toc88211254)

[Burst Sort: 44](#_Toc88211255)

[Intro Sort: 45](#_Toc88211256)

[Tree Sort: 47](#_Toc88211257)

[Shell Sort: 49](#_Toc88211258)

[Cocktail Sort: 52](#_Toc88211259)

[Searching Techniques 55](#_Toc88211260)

[Linear Search 55](#_Toc88211261)

[Binary Search: 56](#_Toc88211262)

[Multiple Filters 57](#_Toc88211263)

[Integration of GUI and Code 58](#_Toc88211264)

[Challenges Faced 65](#_Toc88211265)

[Encoding Error 65](#_Toc88211266)

[Index Out of Bound Error 66](#_Toc88211267)

[Progress Bar Error 66](#_Toc88211268)

[Infinite Loop Error 66](#_Toc88211269)

[Wrong Attributes Error 66](#_Toc88211270)

[Null Object Error 66](#_Toc88211271)

[Things we failed to implement 67](#_Toc88211272)

# List of Figures

[Figure 1: Use case 1 7](#_Toc88211001)

[Figure 2: Use case 2 8](#_Toc88211002)

[Figure 3: Use case 3 9](#_Toc88211003)

[Figure 4: Use case 4 10](#_Toc88211004)

[Figure 5 Insertion Sort Dry Run 31](#_Toc88211005)

[Figure 6 Merge Sort Dry run 40](#_Toc88211006)

[Figure 7 Dry run for Quick Sort 44](#_Toc88211007)

[Figure 8 Dry run for Tree Sort 49](#_Toc88211008)

[Figure 9: Final form for GUI 59](file:///C:\Users\Ushher\OneDrive\Documents\GitHub\CS261F21PID51\Mobile%20and%20Computer%20Accessories%20Scrapper.docx#_Toc88211009)

[Figure 10: Use case for scrapping 60](#_Toc88211010)

[Figure 11: Use case for Start, pause and resume 61](#_Toc88211011)

[Figure 12: Use case for importing data 62](#_Toc88211012)

[Figure 13: Use case for Sorting data 63](#_Toc88211013)

[Figure 14: Use case for searching data 64](#_Toc88211014)

[Figure 15: Use case for Export Data 65](#_Toc88211015)

**List of Tables**

[Table 1: Use case for sorting data 8](#_Toc88211361)

[Table 2: Use case for search 9](#_Toc88211362)

[Table 3: Use case for Export and Import data 10](#_Toc88211363)

[Table 4: Use case for Progress Bar 11](#_Toc88211364)

[Table 5: Time Complexity for Counting Sort 13](#_Toc88211365)

[Table 6: Time complexities for Radix Sort 18](#_Toc88211366)

[Table 7: Time Complexity of Bucket Sort 21](#_Toc88211367)

[Table 8: Time Complexity of Heap Sort 24](#_Toc88211368)

[Table 9: Time complexities for Tim Sort 29](#_Toc88211369)

[Table 10: Time Complexity for Insertion Sort 31](#_Toc88211370)

[Table 11: Time Complexity for Bubble Sort 34](#_Toc88211371)

[Table 12: Time Complexity for Selection Sort 37](#_Toc88211372)

[Table 13: Time Complexity for Merge Sort 40](#_Toc88211373)

[Table 14: Time Complexity for Quick Sort 43](#_Toc88211374)

[Table 15: Time Complexity for Burst Sort 45](#_Toc88211375)

[Table 16: Time Complexity of Intro Sort 47](#_Toc88211376)

[Table 17: Time Complexity of Tree Sort 49](#_Toc88211377)

[Table 18: Time Complexity of Shell Sort 52](#_Toc88211378)

[Table 19: Time Complexity for Cocktail Sort 55](#_Toc88211379)

[Table 20: Time Complexity of Linear Search 56](#_Toc88211380)

[Table 21: Time Complexity of Binary Search 58](#_Toc88211381)

[Table 22 Use case information for scrapping 60](#_Toc88211382)

[Table 23 Use case information for pause, resume and cancel progress 61](#_Toc88211383)

[Table 24 Use case for importing data 62](#_Toc88211384)

[Table 25 Use case for sorting table 63](#_Toc88211385)

[Table 26 Use case for searching items 64](#_Toc88211386)

[Table 27 Use case for exporting data 65](#_Toc88211387)

# 

# Project Description:

A software for scrapping Mobile and Computer accessories from websites. We can search products on different basis like Company, Price, Model etc. with a simple and easy to use interface.

Scrapping data of electronic products from different websites greatly reduces user’s effort who are looking for a product. Also, many different websites have different prices for the same product. This project will help the user to find the optimal product value for money. In addition to that, user can sort and filter products ranging from 0 up to a million. He can start, stop and resume scrapping, any time he wants. This software aims for collecting data from the web and providing you with the best the websites have to offer.

To make things more convenient this software, provide user different sorting algorithms and things like multi-level sorting, that makes it much easier for the user to analyze data. It reduces both effort and time of the user who is looking for products related to mobiles and computers and provide them easy to analyze data and store it on their personal computers. It will also permanently save the data it has pulled, for future use by the consumer. This extracted data can be used for various processes like comparison, analysis and verification upon the client’s needs.

User can also select path to which they want to export data, and similarly can choose a file from which they want to load data. This program will also make use of hybrid algorithms to give the users best time efficiency and accuracy.

## Motivation and Impact:

The motivation of this project was to make an application that provides users with quick and efficient extraction for data. It can be used by a simple user who is looking for an electronic product. Instead of searching products on different websites and manually having to look through each of the pages, he can simply use this application to get faster and more efficient results.

It can also be used by different electronics and related companies to make business decisions on the base of collected data and user feedback about the products. It could be highly impactful, because many Businesses analyze the feedback and such, to predict the upcoming demand and getting to know the idea of the market. Many companies also employ these scrapping techniques to pull data from forums and social media.

# Project Features:

This project has three key components; Web Scrapping, Data Storage and UI.

**Web Scrapping:**

Web scrapping is the extraction of data from a website. This information is then saved and stored into an acceptable file format. In this case, the file format used is .csv. Normally, if someone wanted to access the information of a product, or search a specific product he would have to search through all the different kinds of websites on the internet. Web scrapping helps to overcome this problem by getting data at a much faster rate.

This application scrapes data about Mobile and computer accessories. It will scrap about a million products, giving the user incredible varieties and choices.

**Data Storage:**

The application displays the data scrapped on a UI component, providing Searching and sorting facilities. The user will be provided with the choice to sort data in ascending or descending order.

The data is read from a .csv file, and is displayed in a tabulated form.

Use of sorting algorithms is due to the large number of data set. Inefficient sorting algorithms can take quite a bit of time, which is not optimal for the user.

The following information about the scrapped products is provided:

1. Name
2. Model
3. Manufacturer
4. Price
5. Rating
6. Number of reviews
7. Link of Product on the actual website

These attributes are used as filters, in order to provide searching parameters, and are sorted by different algorithms, providing different efficiencies and time.

# Scrapper Functionalities:

A “scraper” or “scrapper”, is a bot or an automated software that visits HTML pages and websites, and extracts data from its contents. It’s one of the most efficient ways to get data from different websites, and also to take that data to another website or portal. Data scrapping is widely used for:

1. Earning money
2. Research for web
3. Price comparison
4. Selling products from an online site to another consumer

For the purpose of scrapping products from different websites, we built our own scrappers for each particular website. Following were the websites:

1. Amazon
2. Daraz
3. NewEgg

## Functionality:

Each of the scrapper had something in common; it traversed the page and pulled information about each product. It then proceeded to extract the name, manufacturer, model, price, rating and reviews. The attributes were then put into an array, and exported into the file. To traverse the whole category of a product, we get the number of pages from the HTML, and use a loop to traverse the whole category. This made the scrapper even more useful, and we just had to provide the Product name.

The logic behind each of the scrapper is same, the only difference is in the implementation. While one scrapper is fit for scrapping data of “Daraz” website, the other is tuned for the scrapping of the “Amazon” website. It cannot be a case where a scrapper can scrap websites from “Daraz’ and “Amazon”. It is because both websites have different layouts, classes and tags, which leads to a different HTML format altogether.

# User Interface:

The User Interface displays the product attributes/specifications in a tabular form for easy scrolling and reading. There are dropdowns to choose filters, i.e., attributes on the base of which sorting will be implemented. These dropdowns also provide the choice of sorting algorithms for the user. User will enter the path, or the file name (if the file is located in the same folder), to import data. Similarly, he can also export the data into a file and into the system. The user interface also consists of a start, stop, and a resume button, to affect the process of scrapping. The UI also contains a percentage bar, to let the user know how much of the data is scrapped, out of 1 million entities. Overall, the UI is focused on the user’s comfort and to let him know of the processes taking place.

The user interface is designed using PyQt5, which first makes a .UI file and then converts it into aa .py file.

# Collaboration:

## Work done by 2020-CS-107:

* New Egg Scrapper
* Algorithm Report (70%)
* Created Sorting functions
* Final report
* Helped in the integration process

## Work done by 2020-CS-126:

* Amazon Scrapper
* Daraz Scrapper
* Initial proposal Report
* GUI
* Integration of already creatd algorithms with the GUI

# Graphical User Interface:

Use Case # 1 **(Sorting Data)**

|  |  |  |  |
| --- | --- | --- | --- |
| Primary Actor | User | | |
| List of Fields on screen | Field Name | Component | Validation |
| Select Column | Drop down | Drop down includes all choices, no need for validation |
| Sort By | Drop Down | Drop down includes all choices, no need for validation |
| How To Access the use case | The sort by drop down contains the list of algorithms to be selected by the user for sorting the data. The choices include Merge Sort, Insertion Sort, etc. Sorting algorithms have varying efficiencies and time complexities.  The select column drop down consists of the column names, which are actually the attributes of each product,  Like Name, Manufacturer, Model etc. | | |
| Screen Design | https://cdn.discordapp.com/attachments/781497277675208705/896786221366378576/unknown.png  Figure 1: Use case 1 | | |

Table : Use case for sorting data

Use Case # 2(Search)

|  |  |  |  |
| --- | --- | --- | --- |
| Primary Actor | User | | |
| List of Fields on screen | Field Name | Component | Validation |
| Search | Text Box | String Validation for names, int validation for price |
|
| How To Access the use case | Search Method involves using different algorithms to search for the entity typed in the text bar. If it is int, it will be validated so it doesn’t have any string literals. Similarly, for string validation, it will be checked if it has any special characters or numbers in it. It will then proceed to use algorithms to perform the search, showing the preferred product. | | |
| Screen Design | https://cdn.discordapp.com/attachments/781497277675208705/896786221366378576/unknown.png  Figure 2: Use case 2 | | |

Table : Use case for search

Use Case # 3 **(Export data and Import Data)**

|  |  |  |  |
| --- | --- | --- | --- |
| Primary Actor | User | | |
| List of Fields on screen | Field Name | Component | Validation |
| File path (Import) | Text Box | String Validation |
| File path (Export) | Password Field | String validation |
|  | Import data | Button | No validation |
|  | Export Data | Button | No validation |
| How To Access the use case | The text box for export data will either take a file name or a path where the file is to be stored. In the case of a file name, it will save it in the same folder as the program. In the other case, it will save the file at the required destination. The “Export Data” confirms this operation and saves the data into the file.  The same goes for Import data Text box. If there is only file name written, it will look for the file in the same folder as the program. If a path is provided, it will pull the data from the said location. The “Import data” confirms this operation and loads the data onto the Table. | | |
| .Screen Design | https://cdn.discordapp.com/attachments/781497277675208705/896786221366378576/unknown.png  Figure 3: Use case 3 | | |

Table : Use case for Export and Import data

Use Case # 4 **(Progress Bar)**

|  |  |  |  |
| --- | --- | --- | --- |
| Primary Actor | User | | |
| List of Fields on screen | Field Name | Component | Validation |
| Progress | Progress | None |
| Start | Button | None |
|  | Stop | Button | None |
|  | Resume | Button | None |
| How To Access the use case | The progress bar displays the percentage of the entities scrapped. It fills up as the number of entities go larger, reaching 100% when all entities have been scrapped. The Start, Stop and Resume Button transfer the control tho the user, so he can start stop or resume the process of scrapping anytime he wants. | | |
| Screen Design | https://cdn.discordapp.com/attachments/781497277675208705/896786221366378576/unknown.png  Figure 4: Use case 4 | | |

Table : Use case for Progress Bar

# Sorting Techniques:

Different algorithms are used in this project for sorting. Their list and Explanation are given below:

## Counting Sort:

Description:

Counting sort is one of the linear sorting algorithms and is stable (maintains the positions of elements with same key in sorted array as it was in unsorted array). Its time complexity is O (n + k) which is also equivalent to O(n). In this algorithm, first we find the maximum element m from the unsorted array, then create a count array of size m. Now we simply iterate through array and put element n of the unsorted array as an increment of one in the count array at index. After the completion of this loop, we will simply read the indexes of the count array and place the indexes that are not zero in a new array and decrement the count array at that index (we won’t move to next index unless the current count array is 0). This process is repeated for the entire count array and as a result a sorted array is formed.

**Pseudo Code:**

**findRange(array){**

**maximum = array[0]**

**for i in range(0,len(array)):**

**{ if maximum < array[i]:**

**maximum = array[i]**

**minimum = array[0]}**

**for i in range(0,len(array)){**

**if minimum > array[i]:**

**minimum = array[i]}**

**rangeOfArray = maximum - minimum**

**newArray = []**

**for i in range(0,rangeOfArray+1){**

**newArray.append(0)}**

**return newArray,minimum**

**}**

**countSort(array){**

**count , minimum= findRange(array)**

**output = []**

**for i in range(0,len(array)):**

**count[array[i] + abs(minimum)] = count[array[i] + abs(minimum)] + 1**

**i = 0**

**while(i < len(count)):**

**if count[i] > 0:**

**count[i] = count[i] - 1**

**output.append(i - abs(minimum))**

**else:**

**i = i + 1**

**return output**

**}**

Code:

def findRange(array):

    maximum = array[0]

    for i in range(0,len(array)):

        if maximum < array[i]:

            maximum = array[i]

    minimum = array[0]

    for i in range(0,len(array)):

        if minimum > array[i]:

            minimum = array[i]

    if minimum <= 0:

        rangeOfArray = maximum - minimum

    else:

        rangeOfArray = maximum

    newArray = []

    for i in range(0,rangeOfArray+1):

        newArray.append(0)

    return newArray,minimum

def countSort(array):

    count , minimum= findRange(array)

    output = []

    for i in range(0,len(array)):

        if minimum <=0:

            count[array[i] + abs(minimum)] = count[array[i] + abs(minimum)] + 1

        else:

            count[array[i]] = count[array[i]] + 1

    i = 0

    while(i < len(count)):

        if count[i] > 0:

            count[i] = count[i] - 1

            if minimum <=0:

                output.append(i - abs(minimum))

            else:

                output.append(i)

        else:

            i = i + 1

    return output

Time Complexity:

Table : Time Complexity for Counting Sort

|  |  |
| --- | --- |
| **Cases** | **Time Complexity** |
| Best Case | N + r |
| Average Case | N + r |
| Worst Case | N + r |

Proof Of Correctness:

**Loop Invariant** = the size of count array is always equal to the difference between smallest and largest value in the unsorted array and the elements that are not present in the unsorted list with in the range will have 0 at their respective index in the count array

**Initialization:** the count array is initialized with 0 at every index

**Maintenance:** with each iteration the count[element] will increase by one

count[element] = count[element] + 1

**Termination:** When the iteration of unsorted array is over, the count array will contain the count of all elements that were present in the array at their respective count[element] indexes, while the elements that were not present in the array will have a count of zero. Therefore, if we get the indexes of the count array where the count is not zero, we will get the sorted array that we were looking for

Strengths & Weaknesses:

* O(n) time complexity
* Stable sort
* Use less space if we want to sort small numbers
* Use more space if we require to sort large numbers
* Worst time complexity O (n + k)
* Unfit for sorting large databases

Dry Run:

Count Sort

6

7

10

3

1

Here the range is from 1 to 10 and the minimum element is 1

After loop, a count array will be created with respect to the original array containing the count of an element a at count[a] shown in fig below

1

0

1

0

0

1

1

0

0

0

1

Now from this count array, a sorted array will be created

3

6

7

10

1

## Radix Sort:

Description:

Radix sort is a non-comparative (n\*k/d) sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. In this sorting algorithm, solves the issue of high space occupation in counting sort. 10 buckets are created in this method. The elements are repeatedly appended in the bucket of index matching the least significant digit and after each iteration moving forward to next least significant digit. After that, each element will be sorted.

Pseudo Code:

GETNUMBER (num,n)

try

s = str(num)

index = len(s) - n

if(index < 0)

return 0

n = int(s[len(s)-n])

return n

except IndexError

return 0

CHECKBUCKET(bucket)

for i in range(10):

if bucket[i]

return True

return False

COMBINEBUCKET(bucket)

arr = []

for i in range(10):

arr = arr + bucket[i]

return arr

RADIXSORT(array)

bucket = [[] for i in range(10)]

greatest = array[0]

for i in range(len(array)):

if greatest < array[i]

greatest = array[i]

size = len(str(greatest))

for i in range(len(array)):

a = array[i]

a = a%10

bucket[a].append(array[i])

bucket1 = [[] for i in range(10)]

for i in range(1,size+1)

for j in range(10)

if checkBucket(bucket):

while len(bucket[j]) >0

num = bucket[j].pop(0)

a = getNumber(num,i)

bucket1[a].append(num)

for j in range(10)

if CHECKBUCKET(bucket1)

while len(bucket1[j]) >0

num = bucket1[j].pop(0)

a = getNumber(num,i)

bucket[a].append(num)

return COMBINEBUCKETt(bucket)

Code:

RadixSort

def getNumber(num,n):

    try:

        s = str(num)

        index = len(s) - n

        if(index < 0):

            return 0

        n = int(s[len(s)-n])

        return n

    except IndexError:

        return 0

def checkBucket(bucket):

    for i in range(10):

        if bucket[i]:

            return True

    return False

def combineBucket(bucket):

    arr = []

    for i in range(10):

        arr = arr + bucket[i]

    return arr

def radixSort(array):

    bucket = [[] for i in range(10)]

      greatest = array[0]

    for i in range(len(array)):

        if greatest < array[i]:

            greatest = array[i]

    size = len(str(greatest))

    for i in range(len(array)):

        a = array[i]

        a = a%10

        bucket[a].append(array[i])

    bucket1 = [[] for i in range(10)]

    for i in range(1,size+1):

        for j in range(10):

            if checkBucket(bucket):

                while len(bucket[j]) >0:

                    num = bucket[j].pop(0)

                    a = getNumber(num,i)

                    bucket1[a].append(num)

        for j in range(10):

            if checkBucket(bucket1):

                while len(bucket1[j]) >0:

                    num = bucket1[j].pop(0)

                    a = getNumber(num,i)

                    bucket[a].append(num)

    return combineBucket(bucket)

Time Complexity:

Table : Time complexities for Radix Sort

|  |  |
| --- | --- |
| **Cases** | **Time Complexity** |
| Best Case | n\*k/d |
| Average Case | n\*k/d |
| Worst Case | n+2^d |

Proof of Correctness:

Loop invariant: the array is sorted on the basis of (D – i) digit, where D is the number of digits of a number

**Initialization:**

initially i = 0 and the array is sorted on the basis of D which is the least significant digit of a number

**Maintenance:**

after each complete iteration of the loop, the array is again sorted on the basis of D-i with i = i+1

**Termination:**

when i > D the loop will break and the bucket will contain all elements in of the unsorted array in sorted manner

Strengths and Weaknesses:

* It solves the problem of high space occupation as compared to count sort
* Stable sort
* Fast for sorting numbers having small number of digits
* It cannot be used for sorting numbers with decimals
* High value of constant when digits of a number are high

Dry Run:

Let us consider the array as follows

154

54

12

233

98

Now we will create an empty bucket of size 10

Now we will insert the element of array on the basis of right most value:

12

233

154,54

98

Now we will sort on the basis of second last element

233

12

154,54

98

12,54,98

233

154

Now a sorted array is created

12

98

54

154

233

## Bucket Sort:

Description:

Bucket sort is a O(n+k) time complexity sorting algorithm which distribute elements into different buckets and then apply insertion sort on each bucket to sort them. After it, the buckets are merged or combined onto a single array output

Pseudo Code:

INSERTIONSORT(array):

if len(array) <= 0

return array

for i in range(1,len(array))

key = array[i]

j = i - 1

while i >=0 and array[j] > key

array[j + 1] = array[j]

j = j - 1

array[j + 1] = key

return array

def combineBucket(bucket)

array = []

for i in range(10)

array = array + bucket[i]

return array

def bucketSort(array)

bucket = [[] for i in range(10)]

for i in range(len(array))

bucket[int(array[i]\*10)].append(array[i])

for i in range(10):

if bucket[i]:

bucket[i] = insertionSort(bucket[i])

return COMBINEUCKET(bucket)

Code:

bucketSort

def insertionSort(array):

    if len(array) <= 0:

        return array

    for i in range(1,len(array)):

        key = array[i]

        j = i - 1

        while i >=0 and array[j] > key:

            array[j + 1] = array[j]

            j = j - 1

        array[j + 1] = key

    return array

def combineBucket(bucket):

    array = []

    for i in range(10):

        array = array + bucket[i]

    return array

def bucketSort(array):

    bucket = [[] for i in range(10)]

    for i in range(len(array)):

        bucket[int(array[i]\*10)].append(array[i])

    for i in range(10):

        if bucket[i]:

            bucket[i] = insertionSort(bucket[i])

    return combineBucket(bucket)

Time Complexity:

Table : Time Complexity of Bucket Sort

|  |  |
| --- | --- |
| **Cases** | **Time Complexity** |
| Best Case | O(n) |
| Worst Case | O(n²) |
| Average Case | O (n + k) |

Proof Of Correctness:

Proof of correctness using loop invariance is shown below:

Initialization: Prior to the first iteration, k = a, so the subarray X[ a … k-1] is empty and k-a = 0 elements. I = j = 1, no left\_copy[i] and right\_copy[j] contain smallest elements not copied back into X.

Maintenance: Lets consider left\_copy[i] <= right\_copy[j]

Left\_copy[i] -> smallest element not copied into A

X[a … k-1] contain k-a smallest elements.

Now X[k] = Left\_copy[i] therefore, X[a … k] contains k-a+1 smallest elements.

Now i is incremented, k is incremented in the for loop.

This help re-establish the loop invariant.

Termination: k = r+1. By definition of loop invariant. A[a … k-1] which is A[a … r] contains k-a = r-a+1 smallest elements in sorted order

Strengths and Weakness:

* It works best when input is evenly distributed with a time complexity of O (1)
* It is faster than bubble sort because each bucket is sorted separately
* Average time complexity of O (n + k)
* It is not suitable for inputs that are not evenly distributed
* Worst time complexity O(n^2)
* Not applicable to all data types

Dry Run:

Let us consider the array as follows

0.15

0.38

0.12

0.3

0.9

Now we will create an empty bucket of size 10

0.9

0.38,0.3

0.15,0.12

Applying insertion sort in each bucket

0.12,0.15

0.3, 0.38

0.9

Now a sorted array is created by popping out the elements from each bucket

0.12

0.3

0.15

0.38

0.9

## Heap Sort:

Description:

It is an n log n algorithm which is based on binary heap data structure. It divides the array into sorted and unsorted parts and pick the largest element from unsorted array and place it in sorted array at its right location. A heap tree is created using heapify method and then sorting method is applied

Pseudo Code:

HEAPIFYDOWN(array,index,upper)

while True:

left = index\*2+1

right = index\*2+2

if max(left,right) < upper

if array[index] >= max(array[left], array[right])

break

elif array[left] > array[right]

array[index],array[left] = array[left],array[index]

index = left

else:

array[index],array[right] = array[right],array[index]

index = right

elif left < upper:

if array[left] >array[index]

array[left], array[index] = array[index],array[left]

index = left

else:

break

elif right < upper

if array[right] > array[index]

array[right],array[index] = array[index], array[right]

index = right

else:

break

else:

break

HEAPSORT(array)

for i in range((len(array)-2)//2, -1, -1)

heapifyDown(array,i,len(array))

for j in range(len(array)-1,0,-1)

array[j],array[0] = array[0],array[j]

heapifyDown(array,0,j)

array = [2,65,28,12,8,0,183,123]

heapsort(array)

print(array)

Code:

def heapifyDown(array,index,upper):

    while True:

        left = index\*2+1

        right = index\*2+2

        if max(left,right) < upper:

            if array[index] >= max(array[left], array[right]):

                break

            elif array[left] > array[right]:

                array[index],array[left] = array[left],array[index]

                index = left

            else:

                 array[index],array[right] = array[right],array[index]

                 index = right

        elif left < upper:

            if array[left] >array[index]:

                array[left], array[index] = array[index],array[left]

                index = left

            else:

                break

        elif right < upper:

            if array[right] > array[index]:

                array[right],array[index] = array[index], array[right]

                index = right

            else:

                break

        else:

            break

def heapsort(array):

    for i in range((len(array)-2)//2, -1, -1):

        heapifyDown(array,i,len(array))

    for j in range(len(array)-1,0,-1):

        array[j],array[0] = array[0],array[j]

        heapifyDown(array,0,j)

array = [2,65,28,12,8,0,183,123]

heapsort(array)

print(array)

Time Complexity:

Table : Time Complexity of Heap Sort

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

Proof Of Correctness:

The first for loop in the HEAPSORT function creates a max heap using heapifyDown method

Due to which the root of the heap tree is the maximum element of the unsorted array

**Loop invariant:** the first element in the heap is always maximum

**Initialization:** the array is converted into a max heap so that the first element is the maximum. Then j is initialized with length of array

**Maintenance:** with each iteration the first element e.g. array [0] is replaced with array[j] and j is decremented by one. So in the next iteration, heapifyDown method will place the largest element from array (of length 0 to j) to first index and swap it with array[j] followed by decrement in j by one, therefore, maintaining the loop invariant

**Termination:** Loop terminates when j approaches 0. We can see that the maximum element is still at one if consider heap from zero to j, as now it contains only single element. Therefore, the loop invariant remains true after termination

Strengths and Weakness:

* A fast sorting algorithm with best case time complexity of O(n)
* Less memory usage
* Simple and consistent
* Worst case is O(nlogn) which makes it slower than quicksort and mergesort
* Unstable
* Space complexity 1

Dry Run:

154

54

12

233

98

Converting the array in max heap

233

12

154

54

98

Replacing the first element with the last element

98

12

154

54

233

Now creating max heap excluding the last element

154

12

98

54

233

Swapping it with second last element

54

12

98

154

233

Creating max heap excluding last two elements

98

12

54

154

233

Swapping it with third last element

12

98

54

154

233

Again, creating max heap involving only first two elements

54

98

12

154

233

Again swapping

12

98

54

154

233

Now we have a sorted array

## Tim Sort:

Description:

It is an n log n hybrid algorithm derived from merge sort and insertion sort. It uses these algorithms to get best results from them when they are suitable for use. This algorithm uses a minimum interval usually from 32-64. In my case I have used 32 as my interval. If the length of array is lesser than the interval, insertion sort will not implement and only merge sort will take place. If length of array is greater than 32 both insertion sort and mergesort will apply

Pseudo Code:

INSERTION\_SORT(array,left,right):

if right is None

right = len(array)-1

for i in range(left+1,right+1)

key = array[i]

j = i - 1

while j >= left and array[j] > key:

array[j+1] = array[j]

j = j-1

array[j +1] = key

return array

def MERGE(array1,array2)

len1 = len(array1)

len2 = len(array2)

i = 0

j = 0

arr = []

while i < len1 and j < len2

if array1[i] < array2[j]

arr.append(array1[i])

i = i + 1

else

arr.append(array2[j])

j = j + 1

while i < len1

arr.append(array1[i])

i = i + 1

while j < len2

arr.append(array2[j])

j = j + 1

return arr

def TIMSORT(array)

minRun = 24

size1 = len(array)

for i in range(0,size1,minRun)

insertion\_sort(array,i,min((i + minRun -1),(size1-1)))

size2 = minRun

while size2 < size1

for i in range(0,size1,size2\*2)

mid = i + size2-1

end = min((i + size2\*2-1,(size1-1)))

left = array[i:mid]

right = array[mid+1:end+1]

arr = merge(left,right)

size2 = size2 \* 2

return array

Code:

def insertion\_sort(array,left,right):

    if right is None:

        right  = len(array)-1

    for i in range(left+1,right+1):

        key = array[i]

        j = i - 1

        while j >= left and array[j] > key:

            array[j+1] = array[j]

            j = j-1

        array[j +1] = key

    return array

def merge(array1,array2):

    len1 = len(array1)

    len2 = len(array2)

    i = 0

    j = 0

    arr = []

    while i < len1 and j < len2:

        if array1[i] < array2[j]:

            arr.append(array1[i])

            i = i + 1

        else:

            arr.append(array2[j])

            j = j + 1

    while i < len1:

        arr.append(array1[i])

        i = i + 1

    while j < len2:

        arr.append(array2[j])

        j = j + 1

    return arr

def timsort(array):

    minRun = 24

    size1 = len(array)

    for i in range(0,size1,minRun):

        insertion\_sort(array,i,min((i + minRun -1),(size1-1)))

    size2 = minRun

    while size2 < size1:

        for i in range(0,size1,size2\*2):

            mid = i + size2-1

            end = min((i + size2\*2-1,(size1-1)))

            left = array[i:mid]

            right = array[mid+1:end+1]

            arr = merge(left,right)

    size2 = size2 \* 2

    return array

Time Complexity:

Table : Time complexities for Tim Sort

|  |  |
| --- | --- |
| **Cases** | **Time Complexity** |
| Best Case | n |
| Worst Case | N log n |
| Average Case | N log n |

Proof Of Correctness:

Tim sort divides the array into chunks of 32 elements. Then apply insertion sort into these chunks and then use merge sort in these sorted chunks to make a single sorted array

**Loop Invariant:** for each chunk, the subarray[0 to i] is always sorted

**Initialization:** Consider each chunk as separated arrays and subarray[0 to i] of each array. Each sub array is sorted because initially i is equal to 0 and the array containing only one element is considered to be sorted

**Maintainance:** with each iteration a key value (array[i+1]) is compared with the elements of the sorted part and placed in its correct position followed by an increment of i by one. Thus, maintaining the loop invariant

**Termination:** when i approaches the size of original array, the array is completely sorted meaning array[0 to i] is sorted

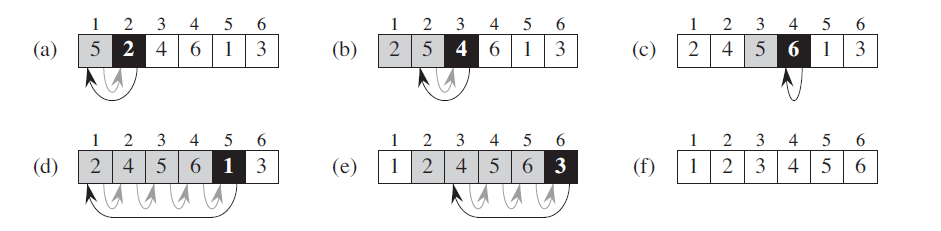
After termination each of the chunks are merged using merge sort into a single sorted array

Strengths and Weakness:

* Optimizes merge sort
* Faster than regular merge sort and quick sort
* Extremely fast for nearly sorted data
* Optimal for sorting real world data
* Worst complexity n log n
* Space complexity n
* Occupy high storage of large number of data

Dry Run:

Same as Insertion Sort



## Insertion Sort:

Description:

Insertion sort is a very simple sorting algorithm. We assume that there are two parts of an array one sorted, and the other unsorted. We pick elements from the unsorted part and place them in sorted part.

Pseudo Code:

For i to length of array:

Key = A[i]

j = i -1

While j >= 0 and A[j] > key

A[j+1] = A[j]

j = j – 1

A[j +1] = key

Code:

def insertion\_sort(array):

    for i in range(0, len(array)-1):

        key = array[i]

        j = i - 1

        while j >= 0 and array[j] > key:

            array[j+1] = array[j]

            j = j-1

        array[j +1] = key

    return array

Time Complexity:

Table : Time Complexity for Insertion Sort

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

Proof Of Correctness:

Loop Invariant:

At the start of each iteration of the ‘For’ loop, the sub-array A [1….j-1] is in sorted order.

**Initialization:**

In i = 2, j = i-1 = 1,

The sub-array becomes A[1], which is a single element. As a single element is always sorted , the loop invariant holds true at initialization

**Maintenance:**

At j, A[1. . . . j -1] is sorted. (the loop invariant)

The while loop then finds the correct position for A[j]. Now A[1. . . . . j] is sorted.

At next iteration of the for loop, j = j+1, so A[1.. .. . . j] is sorted, which means that the loop invariant holds true.

**Termination:**

The loop invariant terminates when j = n +1, i.e., j > n

Which means,

* A[1. . . . . j -1]
* A[1. . . . (n+1) – 1]
* A[1. . . . . n ] is sorted

This proves that insertion sort is correct.

Strengths and Weakness:

* Efficient for small data
* Is stable
* Less use of Memory
* Inefficient for large data sets

Dry Run:

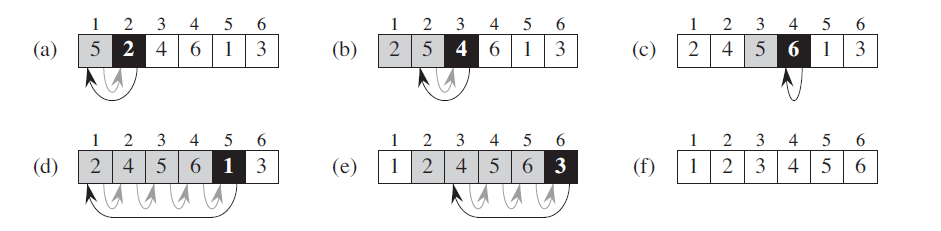


Figure Insertion Sort Dry Run

## Bubble Sort:

Description:

It is another simple algorithm. It traverses the array and swaps two adjacent cells, If they are not in order.

Pseudo Code:

For i to length of array:

For j to length-i-1:

Swapped = False

If array[j] > array[j+1]

swap( list[j], list[j+1] )

swapped = true

if(not swapped):

break

Code:

def Bubble\_sort(array):

    for i in range(0,len(array)):

        swapped = False

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

            if(array[j] > array[j+1]):

                array[j], array[j+1] = array[j+1], array[j]

                swapped = True

        if(not swapped):

            break

    return array

Time Complexity:

Table : Time Complexity for Bubble Sort

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

Proof Of Correctness:

The proof of Correctness for bubble sort is very similar to Selection sort. The only difference is that bubble sort swaps two numbers as it traverses through the array, while Selection sort traverse the array, finds the minimum element, then swaps it with the first element of the unsorted sub array.

Loop Invariant:

The elements in ‘array’[1….i-1], are the smallest of the array in sorted order

**Initialization:**

At i = 0, the array is array is empty, since there are no smallest elements in the array

**Maintenance:**

The sub-array consists of [1.. . . . i-1] elements in sorted order.

At I = i+1, the algorithm keeps traversing the array, swapping any two unsorted element together.

So the array[1…..i-1] becomes array[1……i], which contains elements in sorted order

**Termination:**

The loop invariant terminates when i = n +1, i.e., i > n

Which means,

* array[1. . . . . i -1]
* array[1. . . . (n+1) – 1]

array[1. . . . . n ] is sorted

Strengths and Weakness:

* Good for small data sizes
* Is stable
* Less Use of Memory
* Inefficient for large data sets

Dry Run:

154

54

12

233

98

233

12

154

54

98

Replacing the first element with the last element

98

12

154

54

233

154

12

98

54

233

Swapping it with second last element

54

12

98

154

233

98

12

54

154

233

Swapping it with third last element

12

98

54

154

233

54

98

12

154

233

Again swapping

12

98

54

154

233

Now we have a sorted array

## Selection Sort:

Description:

Selection sort works by finding the minimum element and putting it in the first index. Then it finds the minimum element in the remaining array, and puts it in the second place. It then continues it until the array is sorted.

Pseudo Code:

For i to length of array:

For j to length-i-1:

Swapped = False

If array[j] > array[j+1]

swap( list[j], list[j+1] )

swapped = true

if(not swapped):

break

Code:

def Selection\_sort(Array):

    for i in range(0,len(Array)-1):

        min\_index = i

        for j in range(i+1, len(Array)-1):

            if(Array[j] < Array[min\_index]):

                min\_index = j

        Array[i], Array[min\_index] = Array[min\_index], Array[i]

    return Array

Time Complexity:

Table : Time Complexity for Selection Sort

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

Proof Of Correctness:

Loop Invariant:

The elements in ‘array’[1….i-1], are the smallest of the array in sorted order

**Initialization:**

At i = 0, the array is array is empty, since there are no smallest elements in the array

**Maintenance:**

The sub-array consists of [1.. . . . i-1] elements in sorted order.

At I = i+1, the min\_index is equal to the value in A[i]. The for loop then finds the smallest element in the next sub-array, and if it does, it swaps with the A[i].

In the next iteration , the sub array ‘array’[0…..i] does indeed consist of the smallest numbers in sported order, as now the sub-array becomes [1. . . . i].

**Termination:**

The loop invariant terminates when i = n +1, i.e., i > n

Which means,

* array[1. . . . . i -1]
* array[1. . . . (n+1) – 1]

array[1. . . . . n ] is sorted

Hence Selection sort is correct

Strengths and Weakness:

* Good for small data sizes
* Is stable
* Less Use of Memory
* Inefficient for large data sets

Dry Run:

154

54

12

233

98

12

54

154

233

98

Replacing the first element with the last element

12

154

54

233

98

Now creating max heap excluding the last element

12

98

54

233

154

Swapping it with second last element

12

98

54

154

233

## Merge Sort:

Description:

Merge Sort is a divide and conquer algorithm. It divides the given array into individual elements, and then combines them in sorted order to produce the required output. It is comparison based, so has the lower bound of O (n lg n).

Pseudo Code:

**Merge Sort (arr [], l, r)**

If r > l

1. Find the middle point to divide the array into two halves:

middle m = l+ (r-l)/2

2. Call merge Sort for first half:

Call merge Sort (arr, l, m)

3. Call merge Sort for second half:

Call merge Sort (arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge (arr, l, m, r)

Code:

def merge(array1,array2):

    l1 = len(array1)

    l2 = len(array2)

    i = 0

    j = 0

    array = []

    while i < l1 and j < l2:

        if array1[i] <= array2[j]:

            array.append(array1[i])

            i = i + 1

        else:

            array.append(array2[j])

            j = j + 1

    while i < l1:

        array.append(array1[i])

        i = i + 1

    while j < l2:

        array.append(array2[j])

        j = j + 1

    return array

def mergeSort(array):

    if len(array) <= 1:

        return array

    mid = int(len(array)/2)

    right = array[:mid]

    left = array[mid:]

    right = mergeSort(right)

    left = mergeSort(left)

    arr = merge(left,right)

    return arr

Time Complexity:

Table : Time Complexity for Merge Sort

|  |  |
| --- | --- |
| **Cases** | **Time Complexity** |
| Best Case | O (n lg n) |
| Average Case | O (n lg n) |
| Worst Case | O (n lg n) |

Proof Of Correctness:

Loop Invariant:

Here, three loop are being used. The most important one, that is the core of the merge function is the while loop involving both l1 and l2 arrays So the L.I has two parts:

In the beginning of each iteration, the sub-Array ‘array’[1…… i -1], consists of the smallest element in both the l1 and l2 arrays, in sorted order.

The elements of l1 and l2 are the smallest element of their arrays, not yet added to ‘array’

**Initialization:**

For first part:

At first iteration, the sub array ‘array’ consists of 0 elements i.e., is empty, and has no elements from either l1 or l2.

For second part:

L1[1] and l2[2], are the smallest elements in their respective arrays

**Maintenance:**

For the first part of L.I:

* When the ‘if’ statement evaluates to true, the smallest element ( l1[i] ) will be copied to ‘array’ and therefore, it will contain the smallest element in l1 and l2
* When the ‘if’ statemen evaluated to false the smallest element ( l2[j] ) will be copied to ‘array’ and it will consist of the smallest element in l1 and l2.

For second part:

If array1[i] <= array2[j], I is incremented, and now array1[i] is the smallest element in the respective array. Same goes for when array2[j] < array2[i].

**Termination:**

The loop will terminate when i = l1 +1 or j = l2 + 1. So the sub-array now becomes

‘array’ = [1 . . . .. l1]

OR

‘array’ = [1 . . . .. l2]

which is the sorted array that was the requirement

This proves that Merge sort is correct.

Strengths and Weakness:

* Constant time for all lengths
* Is stable
* Uses more memory
* Constant time even for small inputs
* Fragile

Dry Run:

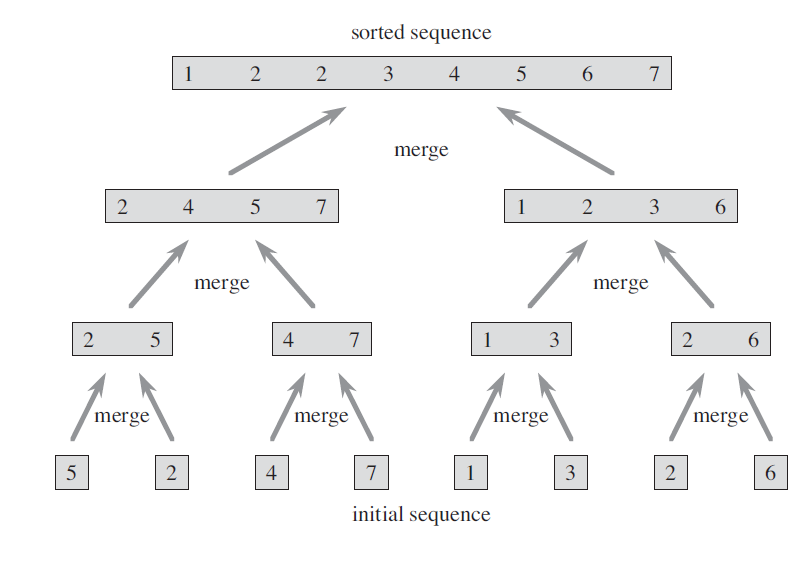


Figure Merge Sort Dry run

## Quick Sort:

Description:

Quicksort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways. Here we picked pivot as the last element. It is also a comparison-based algorithm, hence lower bound by the time complexity n log (n).

Pseudo Code:

quicksort (arr, low, high)

{

if (low < high)

{

pi = partition ( arr, low, high);

quicksort (arr, low, pi - 1); // Before pi

quicksort (arr, pi + 1, high); // After pi

}

}

partition (arr[], low, high)

{

pivot = arr[high];

i = (low - 1)

for (j = low; j <= high- 1; j++)

{

if (arr[j] < pivot)

{

i++; // increment index of smaller element

swap arr[i] and arr[j]

}

}

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

return (i + 1)

Code:

def partition(array, low , high):

    pivot = array[high]     # the index around which array will be sorted

    i = low - 1             # - 1 is to separate base case in the function "quickSelect", to identify if the pivot has an array to it's left or right

    for j in range(low, high):      #loop is to place the pivot in it's right position within the array

        if(array[j] < pivot):

            i += 1

            temp = array[i]

            array[i] = array[j]

            array[j] = temp

   #if the loop finds no suitable position, meaning the pivot is less than all other elements in the array

temp = array[i+1]

    array[i+1] = array[high]

    array[high] = temp

    return i + 1

def QuickSort(array, high, low):

    if (low < high):

        pi = partition(array, low, high)

        left = []

        left.append(QuickSort(array, low, pi - 1))

        right = []

        right.append(QuickSort(array, pi + 1, high))

        return left.append(right)

Time Complexity:

Table : Time Complexity for Quick Sort

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

Proof Of Correctness:

The main thing to prove is the loop in partition.

The loop invariant has three parts:

At the start of each new iteration, i = low -1 and j = low

1. If low <= k <= i, array[k] <= pivot
2. If i+1 <= k <= j-1, array[k] > pivot
3. If k = high, then array[k] = pivot

**Initialization:**

In the beginning of iteration, i= low-1 and j = low, the conditions hold as follows:

1. If low<= k <= low-1, then array[k] <= x, true bcz no value of k satisfies the equation
2. If low <= k <= low-1, then array[k] > pivot, true bcz no value of k satisfies the equation
3. If k = high, the array[k] = pivot, but since no changes have been made to pivot, this condition is also true

**Maintenance:**

Let’s assume that at iteration j the conditions are true

Then,

If A[j] <= pivot then A[j] will be swapped with the first element of the right sub-array and the index of the last element of left sub-array is increased by one(i=i+1)

Hence at the iteration j+1, the following conditions hold:

1. If low <= k <= i, array[k] <= pivot
2. If i+1 <= k <= j-1, array[k] > pivot
3. If k = high, then array[k] = pivot

If A[j] > pivot , the only change is the last index of the right sub array and the conditions remain valid because the last index is greater than pivot

**Termination:**

At Termination, j = high and therefore the array ‘array’ has been partitioned into three sub-arrays, those that are less than the pivot, those that are greater than the pivot, and those that are equal to it.

Strengths and Weakness:

* No excess memory usage
* Takes average of n log (n) time
* Regarded as best due to its efficiency
* Has worst case of O (n^2)

Dry Run:

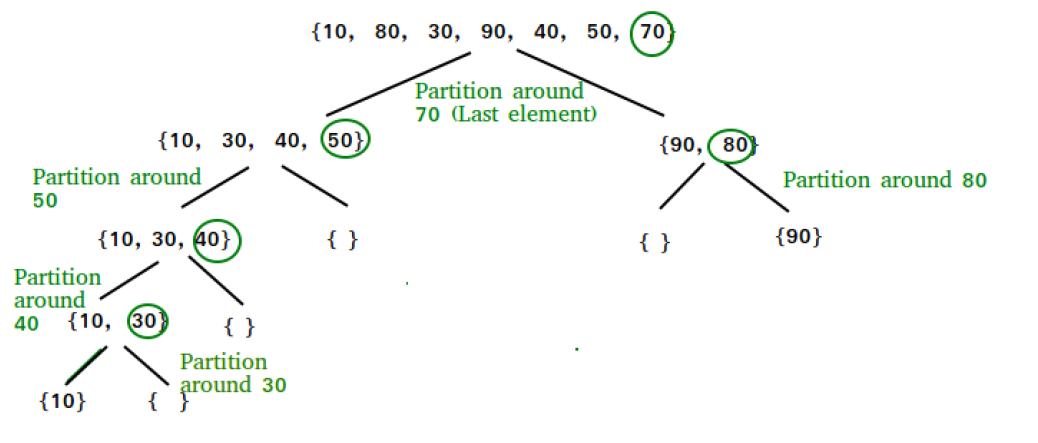


Figure Dry run for Quick Sort

## Burst Sort:

Description:

Burst Sort is an efficient Algorithm, used to sort strings in Burst tries, which itself is another data Structure. It shows better asymptotic behavior than any other algorithm for sorting strings. It is not a comparison-based algorithm, and hence is extremely fast. It has much better Time Complexity than O (n log(n)), it itself having O(n).

Pseudo Code:

-----------

Code:

------------

Time Complexity:

Table : Time Complexity for Burst Sort

|  |  |
| --- | --- |
| **Cases** | **Time Complexity** |
| Best Case | O (wn) |
| Average Case | O (wn) |
| Worst Case | O (wn) |

Proof Of Correctness:

Just as in Binary Search Tree, burst tries store strings in an ordered or a near-ordered fashion. The simple In-order Traversal of the burst tries data structure yields the elements in a sorted order. It uses the same memory as a Binary Search Tree, and can yield much faster results than other methods to store strings in.

Strengths and Weakness:

* Extremely Fast
* Has better time complexity even over Large Data Sets
* Requires another data Structure
* Memory usage is high
* Complex Algorithm

Dry Run:

--------------

## Intro Sort:

Description:

It is hybrid algorithm (n log n) that starts with quick sort, then switch to heap sort at certain level and then to insertion sort. It uses these algorithms to get best results from them when they are suitable for use. If the input is less than a certain definite amount, Insertion sort is used. If the array gets large, we use heap Sort. If the looping or recursion of the quick sort function gets too big, we switch to Quick sort.

Pseudo Code:

Intro Sort (Array):

If length (array) <= 30:

Return Insertion Sort (Array)

Else if (recursion is not deep):

Return Heap Sort (Array)

Else:

Return Quick Sort (Array)

Code:

def Intro\_sort(Array, depth\_limit):

    if(len(Array) <= 30):

        return insertion\_sort(Array)

    elif(depth\_limit == 0):

        return Heap\_sort(Array)

    else:

        return Quick\_sort(Array)

Time Complexity:

Table : Time Complexity of Intro Sort

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

Proof Of Correctness:

This is a hybrid algorithm that calls either insertion sort, Heap sort, or Quick sort depending upon the input. So, if we prove that each of the algorithms is correct, we can prove that Intro Sort is also correct. Since we have proven that Insertion, Heap and Quick sort are correct algorithms, hence this algorithm is also correct.

Strengths and Weakness:

* Fast average and best worst time complexity (n log n)
* Combines the power of Insertion sort, Quick Sort and Heap Sort
* Not stable
* Has the weaknesses of Quick sort if it is chosen

Dry Run:

Same as Heap Sort (page 24), Insertion sort (Figure 5) and Quick Sort (Figure 7)

## Tree Sort:

Description:

A tree sort is a (n log n) algorithm that builds a binary search tree from the elements to be sorted, and then traverses the tree so that the elements come out in sorted order. Tree sorting is a fast sort process. To traverse the binary tree. It first traverses the left subtree, and then right tree. It finds the smallest element at the left sub-tree, starts from there, and ends at the largest element in the right sub-tree. It moves up, from the smallest element, and does not traverse from the top-down, from the node.

Pseudo Code:

inorderTraversal(self, node):

if (self.root == None):

return;

else:

if (node.left != None):

inorderTraversal(node.left)

if (node.right != None):

self.inorderTraversal(node.right);

Code:

def inorderTraversal(self, node):

    #Check whether tree is empty

    if(self.root == None):

        print("Tree is empty")

        return

    else:

        if(node.left != None):

            self.inorderTraversal(node.left)

            print(node.data)

        if(node.right!= None):

            self.inorderTraversal(node.right)

Time Complexity:

Table : Time Complexity of Tree Sort

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

Proof Of Correctness:

Since Tree sort is in reality just the traversing of the binary tree, no proof of Correctness is needed. This algorithm has traversal time complexity as n log (n), due to the properties of the binary tree, that facilitate in faster traversal. However, this comes at this cost of slower insertion and deletion.

Strengths and Weakness:

* Fast Algorithm
* Changes are easy to do
* Simple Algorithm
* Has worst run time if the tree is already sorted

Dry Run:

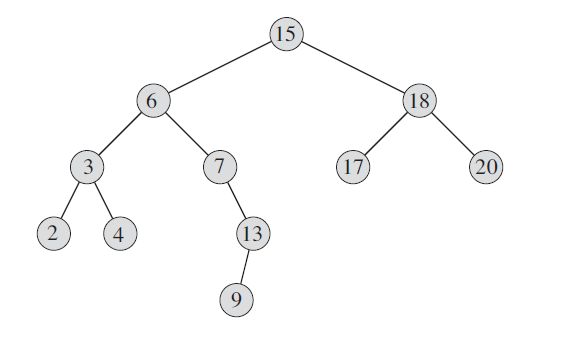


Figure Dry run for Tree Sort

4

20

3

2

18

15

13

6

7

9

17

## Shell Sort:

Description:

Shell sort is a variant of insertion sort. Instead of Doing insertion sort of N numbers, we divide the N numbers into h gaps, and move the value to be inserted across h gaps. This takes less computational power and has improved time complexity than traditional Insertion sort.

Pseudo Code:

Shell sort(array):

Gap = len(array) / 2

While gap > 0:

I= 0

J =0

While j < len(array) :

If(array[i] > array[j])

Swap(array[i], array[j])

I++

J++

K = i

While(k – gap > -1):

If array[k-gap] > array[k]

Swap(array[k-gap], array[k])

K- -

Gap = gap /2

Return array

Code:

def Shell\_sort(array):

    gap = len(array) // 2

    while gap > 0:

        i = 0

        j = gap

        while (j < len(array)):

            if (array[i] > array[j]):

                array[i],array[j] = array[j],array[i]

            i = i + 1

            j = j + 1

            k = i

            while (k - gap > -1):

                if array[k - gap] > array[k]:

                    array[k-gap], array[k] = array[k], array[k-gap]

                k = k - 1

        gap = gap // 2

    return array

Time Complexity:

Table : Time Complexity of Shell Sort

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

Proof of Correctness:

Same as Insertion sort

Strengths and weaknesses:

* Efficient for small data sets
* Faster than bubble sort and insertion sort
* Better time complexities
* Complex and difficult
* Not stable

Dry Run:

154

54

12

233

98

54

154

12

233

98

54

154

233

12

98

54

98

12

233

154

12

54

54

154

233

## Cocktail Sort:

Description:

Similar to shell Sort, cocktail sort is also a variation of another algorithm, Bubble Sort. Instead of traversing the array in one way like bubble sort, it traverses the array back and forth, in one iteration. It is efficient than bubble sort, and has improved time complexity.

Pseudo Code:

Cocktail sort(array)

Swapped = true

Start = 0

End = Array.len – 1

While(swapped):

Swapped = false

For I = start to end:

If(array[i] > array [i+1]):

Swap(array[i], array[i+1])

Swapped = true

If(! swapped):

Break

Swapped = false

End - -

For I = end -1 to start -1 or -1:

If(array[i] > array[i+1]):

Swap(array[i], array[i+1])

Swapped = true

Start ++

Return array

Code:

def cocktail\_sort(array):

    swapped = True

    start = 0

    end = len(array) - 1

    while (swapped == True):

        swapped = False

        for i in range(start, end):

            if (array[i][1] > array[i + 1][1]):

                array[i], array[i + 1] = array[i + 1], array[i]

                swapped = True

        if (swapped == False):

            break

        swapped = False

        end = end-1

        for i in range(end-1, start-1, -1):

            if (array[i][1] > array[i + 1][1]):

                array[i], array[i + 1] = array[i + 1], array[i]

                swapped = True

        start = start + 1

    return array

Time Complexity:

Table : Time Complexity for Cocktail Sort

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

Proof of Correctness:

Same as Bubble sort:

Strengths and weaknesses:

* Stable
* Performs batter than Bubble sort
* Sorting is in place
* Inefficient time complexities
* Only marginal increase in performance than bubble sort

Dry Run:

154

54

12

233

98

154

54

12

98

233

12

54

154

98

233

12

54

98

154

233

12

98

54

154

233

Searching Techniques**:**

Searching algorithms are divided into two main categories:

1. Sequential Search
2. Interval Search

Different algorithms are used in this project, details as follows:

Linear Search**:**

Description:

Linear search is a very simple algorithm. It basically traverses the whole array to look for the required element. If it finds one, it returns the said value. We perform this search using iteration methods (i.e., using loops) or by the recursive method.

Pseudo Code:

For i = 1 to A.length

If(A[i] == value):

Return A[i]

Code:

def Linear\_Search(array, val):

    for i in range (0,len(array)-1):

        if(array[i] == val):

            return array[i]

    return None

Time Complexity:

Table : Time Complexity of Linear Search

|  |  |
| --- | --- |
| **Cases** | **Time Complexity** |
| Best Case | O (1) |
| Average Case | O (n/2) |
| Worst Case | O (n) |

## Binary Search:

Description:

This is also another simple algorithm. The procedure is to keep dividing the array in half, comparing the value to be found with the middle element. If the value is less, we perform binary search on the left sub array, if not then on the right subarray.

Pseudo Code:

BinarySearch(A,start, end, val):

If (len.A >= 1):

Mid = end / 2

If (A[mid]) == val:

Return mid

Else if(A[mid] > val):

BinarySearch(A, 0, mid-1, val)

Else if(A[mid] < x):

BinarySearch(A, mid+1, len.A, val)

Else:

Return None

Code:

def binarySearch (array, x, high ,low):

    if low >= 1:

        print("In if")

        mid = low + (high - low)//2

        # If found at mid, then return it

        if array[mid] == x:

            return mid

        # Search the left half

        elif array[mid] > x:

            return binarySearch(array, x, low, mid-1)

        # Search the right half

        else:

            return binarySearch(array, x, mid + 1, high)

    else:

        print("In else")

        return None

Time Complexity:

Table : Time Complexity of Binary Search

|  |  |
| --- | --- |
| **Cases** | **Time Complexity** |
| Best Case | O (1) |
| Average Case | O (n lg n) |
| Worst Case | O (n lg n) |

Multiple Filters**:**

Multiple filters are used to sort and display the products based on different parameters. In this case, the parameters are the product specifications such as Name, Manufacturer, price etc., (does not include links). These filters are there for the convenience of the user. For example, if the user wants to sort and view the products based on the highest priced, he can choose the column of price and choose the sorting algorithm as he wishes. The table will then display the products sorted on the basis of price.

Similarly, if the user wants to separate the products on the basis of reviews or ratings, he can choose the similar method and view the products on the basis of said parameters. This can be useful, if the intended audience is a business company, and they can know the demand and current trend of the market, as it was discussed in motivation and impact.

Links are not chosen as parameters because it just points to the address to the products web page

to buy the said product.

Integration of GUI and Code**:**

First off, we have the Sorting Section. This section contains two drop boxes, ‘Sort by’ and ‘Select Column’. The “Sort by” column presents choices for the user to choose the sorting algorithms. The column which he chooses in the “Select column”, will have the sorting algorithm applied. Behind the GUI, the dropdown chooses the sorting algorithm and the column is passed as a parameter. The sorting algorithm sorts, and passes the resultant array on to the table, which has the code to display the passed array.

Secondly, the Search Section. It consists of a Search Button and a Textbox titled “Search” This section has the code to search the string or integer input.

In the data set of products. It implements two searching algorithms, Linear Search and Binary Search. We would provide the user with an interesting option. If the user writes the word “Scrap’ + name, the program will start to scrap product with the given name, from the websites. If the user writes the word “Find” + name, the program will start to search the given name in the data set, and provide the results.

Then we have the Save Data portion, which consists of two Textboxes, titled “Import” and “Export” and two buttons with the same titles. These text boxes have the implementation to save and extract data from files or files at another locations. The “Import” uses a function named “with\_open()”, that opens the file and reads it. It then puts the data into a tabular form. The opposite if it happens with the Export Data, which puts the data in the table into a file and exports the file with the name typed into the Text box.

Finally, we have the progress bar and the Table. The progress bar shows the percentage of entities scrapped. The implementation works by incrementing a specific percentage of progress bar, when the loop is executed a specific number of times. The table is then used to call functions that displays the product attributes. The buttons ‘Start’, ‘Stop’ and ‘Resume’ are used to control whether you want the entities to be scrapped or not, and perform the functions indicated by their names.

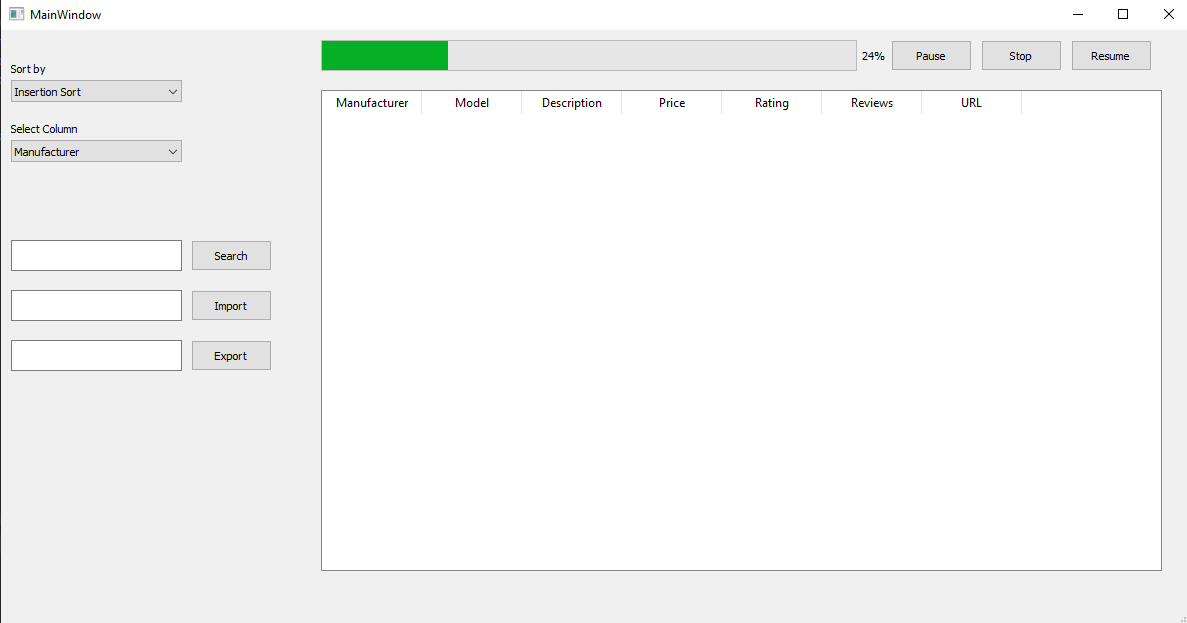
The final form of GUI is below:

Figure : Final form for GUI

**Use cases:**

**Use Case 1:**

Table Use case information for scrapping

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Scrap | | |
| Primary Actor | User | | |
| List of Fields on screen | Field Name | Component | Validation |
| Search Field | Text Box | None |
| Search Button | Button | None |
| ImportField | Text Box | Valid file in the same directory |
| Import Button | Button | None |
| Export Field | Text Box | None |
| Export Button | Button | None |
| Select Column | Drop down | Drop down includes all choices, no need for validation |
| DataTable | Table widget | None |
| Filter Button | Button | None |
| StopButton | Button | None |
| PauseButton | Button | None |
| ResumeButton | Button | None |
| Progress Bar | Progress Bar | None |
| Sort By | Drop Down | Drop down includes all choices, no need for validation |
| How To Access the use case | Type ‘scrap’ and then the name of the product you want to scrap in the search Field and then press search | | |
| Screen Design | Figure 10: Use case for scrapping | | |

**Use Case 2:**

Table Use case information for pause, resume and cancel progress

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Pause, Resume, and Cancel | | |
| Primary Actor | User | | |
| List of Fields on screen | Field Name | Component | Validation |
| Search Field | Text Box | None |
| Search Button | Button | None |
| ImportField | Text Box | Valid file in the same directory |
| Import Button | Button | None |
| DataTable | Table widget | None |
| Filter Button | Button | None |
| Export Field | Text Box | None |
| Export Button | Button | None |
| Select Column | Drop down | Drop down includes all choices, no need for validation |
| StopButton | Button | None |
| PauseButton | Button | None |
| ResumeButton | Button | None |
| Progress Bar | Progress Bar | None |
| Sort By | Drop Down | Drop down includes all choices, no need for validation |
| How To Access the use case | While scraping is in the progress you can pause, resume and stop the scrapping progress as you will by pressing the respective buttons | | |
| Screen Design | Figure 11: Use case for Start, pause and resume | | |

**Use Case 3:**

Table Use case for importing data

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Import data | | |
| Primary Actor | User | | |
| List of Fields on screen | Field Name | Component | Validation |
| Search Field | Text Box | None |
| Search Button | Button | None |
| ImportField | Text Box | Valid file in the same directory |
| Import Button | Button | None |
| Export Field | Text Box | None |
| DataTable | Table widget | None |
| Filter Button | Button | None |
| Export Button | Button | None |
| Select Column | Drop down | Drop down includes all choices, no need for validation |
| StopButton | Button | None |
| PauseButton | Button | None |
| ResumeButton | Button | None |
| Progress Bar | Progress Bar | None |
| Sort By | Drop Down | Drop down includes all choices, no need for validation |
| How To Access the use case | You can import data from a csv file by entering the file path in the importField and then press the import button, your csv file will be displayed on the table | | |
| Screen Design | Figure 12: Use case for importing data | | |

**Use Case 4:**

Table Use case for sorting table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Sorting | | |
| Primary Actor | User | | |
| List of Fields on screen | Field Name | Component | Validation |
| Search Field | Text Box | None |
| Search Button | Button | None |
| ImportField | Text Box | Valid file in the same directory |
| Import Button | Button | None |
| Export Field | Text Box | None |
| Export Button | Button | None |
| DataTable | Table widget | None |
| Filter Button | Button | None |
| Select Column | Drop down | Drop down includes all choices, no need for validation |
| StopButton | Button | None |
| PauseButton | Button | None |
| ResumeButton | Button | None |
| Progress Bar | Progress Bar | None |
| Sort By | Drop Down | Drop down includes all choices, no need for validation |
| How To Access the use case | After the data has printed on the table, you can select the attribute on the basis of which sorting will take place and select any sorting algorithm from sortComboBox and then hit filter, your table will be sorted | | |
| Screen Design | Figure 13: Use case for Sorting data | | |

**Use Case 5:**

Table Use case for searching items

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Search | | |
| Primary Actor | User | | |
| List of Fields on screen | Field Name | Component | Validation |
| Search Field | Text Box | None |
| Search Button | Button | None |
| ImportField | Text Box | Valid file in the same directory |
| Import Button | Button | None |
| Export Field | Text Box | None |
| Export Button | Button | None |
| DataTable | Table widget | None |
| Filter Button | Button | None |
| Select Column | Drop down | Drop down includes all choices, no need for validation |
| StopButton | Button | None |
| PauseButton | Button | None |
| ResumeButton | Button | None |
| Progress Bar | Progress Bar | None |
| Sort By | Drop Down | Drop down includes all choices, no need for validation |
| How To Access the use case | After displaying the table, you can search a sub string in the table just by entering the sub string in the searchField (without prefixing it with scrap) and then hit search, it will display you rows that contain you sub string. After the pressing the search button, the text on the button will change to ‘Back’ and if you wish to see the original table, you can simply press it and the original table will be displayed | | |
| Screen Design | Figure 14: Use case for searching data | | |

**Use Case 6:**

Table Use case for exporting data

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Export data | | |
| Primary Actor | User | | |
| List of Fields on screen | Field Name | Component | Validation |
| Search Field | Text Box | None |
| Search Button | Button | None |
| ImportField | Text Box | Valid file in the same directory |
| Import Button | Button | None |
| Export Field | Text Box | None |
| Export Button | Button | None |
| DataTable | Table widget | None |
| Filter Button | Button | None |
| Select Column | Drop down | Drop down includes all choices, no need for validation |
| StopButton | Button | None |
| PauseButton | Button | None |
| ResumeButton | Button | None |
| Progress Bar | Progress Bar | None |
| Sort By | Drop Down | Drop down includes all choices, no need for validation |
| How To Access the use case | You can export data by entering the path + file name, or simply file name top export in csv to that path or current directory respectively | | |
| Screen Design | Figure 15: Use case for Export Data | | |

Challenges Faced**:**

Encoding Error**:**

During scrapping, there was a very nerve-wracking error, which was the Encoding Error. It basically happened because of the different encoding techniques employed by HTML pages, that was not the encoding the file used. Due, to this error, the system refused to save or display the string and terminated the program. It was tackled by encoding the HTML content into UTF-08, which was then saved into file.

Index Out of Bound Error**:**

With the use of lists and arrays, this error was unavoidable. It happens when the array index we try to access is out of the confines of the array. This occurred mostly in loops. It was solved by adjusting the loops conditions, so that it stops at a value larger than the index.

Progress Bar Error**:**

This error caused the program to crash whenever the progress bar reached 100%. This was most likely due to the loop iterating more than it should, causing an index out of bound hence crashing the program. It was fixed by checking the loop conditions and changing them accordingly.

Infinite Loop Error**:**

This caused the program to run endlessly. Fortunately, python has a built-in function to check whether the program is in an infinite loop or recursion, and terminates the program. This is, again, due to invalid loop conditions, and is solvable by adjusting the said conditions.

Wrong Attributes Error**:**

Often times, in the content of an HTML page, there are also Ads in the same container as the items. When a scrapper runs, it does not distinguish between an Ad and the item. When it tries to pull the attributes of an Ad, it doesn’t find anything and returns Null. This causes the exception of Wrong attributes. It is also partly the cause of above-mentioned Encoding error, because sometimes ads contain such strings. This was tackled by applying if conditions into the scrapper, so that if it finds such strings it knows what to do.

Null Object Error**:**

This error was the source of many problems. It happened when we tried to access a Null object or variable. It mostly happened in conjunction with Wrong Attributes Error and often displayed the statement “Null type object has no attribute str” or “Null type object not sub-scriptable”.

It was difficult to deal with it, as often times we do not know the source of the null pointer. Only by careful reading and proof checking the program can this error be resolved.

Things we failed to implement**:**

**Multi Sorting:**

Due to shortage of time and the pressure of midterm exam, we couldn’t implement multiple sorting. Although, we did try to design it but were unable to implement it because of bugs and it also hindered with the normal functioning of the program so we decided to remove it

**Some Sorting Algorithms:**

Non-comparison-based sorting algorithms like count sort, radix sort and bucket sort were not applicable to the way we had planned to sort, so we didn’t implement those. We designed Tim sort but it was only working for a few sets of input, so we removed it. Similarly, we also didn’t implement intro sort, burst sort, and binary insertion. Instead of those we implemented Bubble sort, shell sort, and cocktail sort

Integration of Daraz and New Egg Scraper:

We did make the scraper for these websites but they had following serious issues:

* Most of the material that the website offer to a search was irrelevant or undesired
* Too many ads which cause some unwanted products to appear in the csv file
* Captcha error was the main issue due to which we had to abandon these scrapers