

**Electronic Products Scrapped**

**Project ID: 42**

**Submitted By:**

**Muhammad Talha 2020-CS-39**

**Submitted To:**

**Dr. Samyan Qayyum Wahla**

**CS-261 Data Structure & Algorithms**

**Department of Computer Science**

**Executive Summary**

Project will retrieve data of electronic products from different websites of different attributes. There will be seven attributes of entity (Name, Price, Discount, Ratings, Country, Reviews, Ranking, Brand). On User interface all these attributes will be shown in columns. One attribute will be in one column and column contains data of entity according to their name. In this way there will be total seven columns each column containing data according to their attribute name.

Searching filters will be shown on each column. These filters will be in dropdown. Dropdown contain names of different searching algorithms. User can search data from any column using any filter and according to any searching algorithm which will be in dropdown box.

Also, dropdowns will be shown on each column for sorting algorithms. Different name of sorting algorithm will be in dropdown. User can sort data according to any column by using their dropdown and using any sorting algorithm which he/she want. In bottom there will be time showing that which algorithm take how much time for sorting the data.

Progress bar and button will be on screen at bottom. Progress bar shows that how much percentage of data is scrapped. If user click on start button scraping of entities will start then this button will become pause button. If user click on pause scraping will pause then this button will become resume button by clicking again on this again scraping will start.

**Business need for the Project**

This project will allow users to see the product details on same page. They can see what rating of the product is and reviews of products as well as their price. And they can sort products data according to any attribute.

**End User Product**

Product domain is the electronics products which is for sale on different websites. People search on websites for their needs. This website will help them to see details of product. End user of the product will be civilians. This will be effective for civilians. This will be able them to take better decision for buying electronic products.

**Motivation of Project**

I made this project for people’s benefits. Civilians can gain much benefits from this for electronic products. They can see products of different brands, prices, ratings, installments available or not, discount percentage. This will help them to take better decision that which product of which company will be better for them.

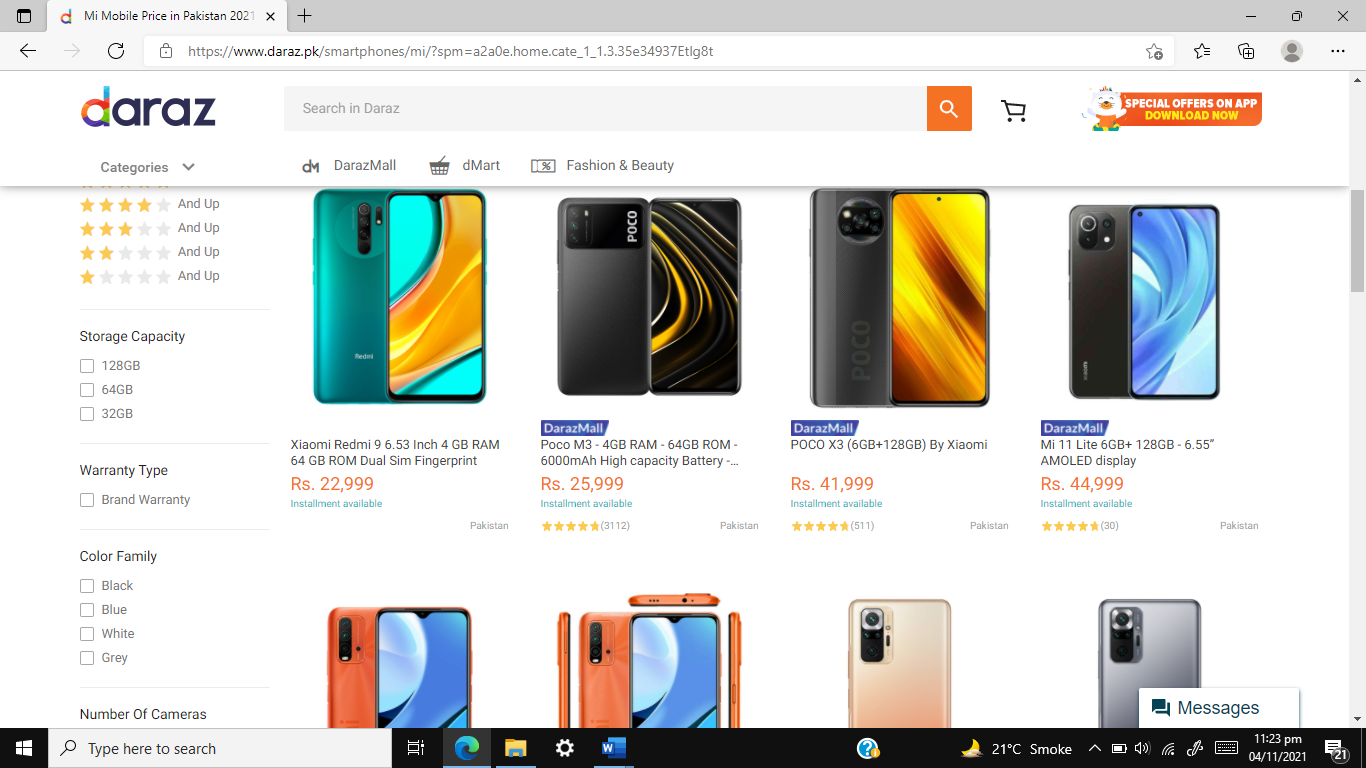
**Scraping**

**Attributes to Scrap**

|  |  |  |
| --- | --- | --- |
| **Name** | **Data Type** | **Description** |
| Name | string | Specifies name of the product |
| Rating | string | Shows how much people liked this product |
| Brand | string | Specifies Name of the brand of product |
| Price | integer | Product price will be points |
| Discount | string | How percentage discount is on the product |
| Installment | string | On The Product installment is available or not |
| Country | integer | In which country product is available |

Website from which would be scrap data is **daraz.pk.** Seven attributes of a product will be scrapped. These are above seven attributes. Below is the sample of website which will be used for scraping.

**Sample Scraping Source**



**Sorting Algorithms**

**Insertion Sort**

**Description**

In this algorithm an element of array is picked from start and compared it with other element until it goes to its right position. Similarly, this procedure follows from all elements of array and so array is sorted.

**Pseudo Code**

def insertionSort( A ):

for i=2 to length.Arrray

key = A[i]

j= i-1

while j>=0 and key < A[j]

A[j+1] = A[j]

J = j-1

A[j+1] = key

Return Array

**Python Code**

    def getSortedArray(self , Array , cloumn , asc):

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

            key = Array[i].data[cloumn]

            keys = Array[i]

            j = i-1

            while j>=0 and ((key<Array[j].data[cloumn]) == asc):

                Array[j+1] = Array[j]

                j = j-1

            Array[j+1] = keys

        return Array

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Best Case **O(n)** | Worst Case **O(n^2)** | Average Case **O(n^2)** |
| When array is already sorted only for loop will execute of length n and while loop no run so only n times operation executes. | When given array is in reverse sorted order outer for loop will execute n times and inner loop will execute n time for one value of n so whole operations will be O(n^2). | When given array is in unsorted not fully reversed sort time will also about O(n^2). |

**Strengths**

* It is very effective when array elements less than 26.
* It sorts array in less times when array is already is near to sort.
* It is very simple to understand.

**Weakness**

* It takes more time when input is large.
* It also takes more time when input array is in near to in complete unsorted order.
* It does not perform well like other sorting algorithms.

**Proof of Correctness**

**Loop Invariant**

At every step index I all values in subarray are sorted.

* **Initialization**

Array start with the first element in the subarray will be in sorted order.

* **Maintenance**

When loop will execute on index, I then array at position I will be sorted and at the next iteration of the loop at the index I array will obviously sorted at position I Left side of I whole array will be sorted.

* **Termination**

Loop will terminate when I will be on last index of the array and at that time whole array will sorted.

**Dry Run**

Input

|  |  |  |
| --- | --- | --- |
| 2 | 10 | 5 |

|  |  |  |
| --- | --- | --- |
| 2 | 10 | 5 |
| 2 | 10 | 5 |
| 2 | 10 | 10 |
| 2 | 10 | 10 |
| 2 | 5 | 10 |

Trace Table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| i | list[i] | key | j | list[j] | list[j+1] |
| 1 | 10 | 10 | 0 | 2 | 10 |
| 2 | 5 | 5 | 1 | 10 | 5 |
| 2 | 5 | 5 | 1 | 10 | 10 |
| 2 | 5 | 5 | 0 | 10 | 10 |
| 2 | 5 | 5 | 0 |  | 5 |

Output:

|  |  |  |
| --- | --- | --- |
| 2 | 5 | 10 |

**Selection Sort**

**Description**

In this algorithm first smallest element is picked form array and placed first then second smallest at second and so on. In this whole array sorted.

**Pseudo Code**

def selectionSort(A):

for i=1 to Length.A-1

index = i

for j=i+1 to Length.A:

if A[j] < A[index]:

index = j

if index!=i:

temp = A[index]

A[index] = A[i]

A[i] = temp

Return A

**Python Code**

    def getSortedArray(self , Array , column , asc):

        for x in range(0 , len(Array)-1):

            index = x

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

                if (Array[y].data[column] < Array[index].data[column]) == asc:

                    index = y

            if index != x:

                temp = Array[index]

                Array[index] = Array[x]

                Array[x] = temp

        return Array

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Best Case **O(n^2)** | Worst Case **O(n^2)** | Average Case **O(n^2)** |
| When array is already sorted for loop will execute of length n and inner loop will also execute weather array is sorted or not. | When given array is in reverse sorted order outer for loop will execute n times and inner loop will execute n time. | When given array is in unsorted not fully reversed sort then also both nested for loop will execute for n time. |

**Strengths**

* It is effective for small number of lists.
* It is easy to understand.
* It is an in-place sorting algorithm no additional memory is required for containing lists.

**Weakness**

* It is much time consuming for large number of inputs.
* It takes same time when array is initially sorted or unsorted order.

**Proof of Correctness**

**Loop Invariant**

At every index I total I values in subarray are sorted.

* **Initialization**

When value of outer loop index will be one then in subarray first smallest element will be in their right position.

* **Maintenance**

When value of outer loop index will be 2 second smallest element of array will also become at their right position in subarray so in this way whole array will be sorted.

* **Termination**

When loop will terminate index will be at last element this will also at right position so at termination of outer loop whole subarray will be sorted.

**Dry Run**

Input:

|  |  |  |
| --- | --- | --- |
| 2 | 10 | 5 |

Trace Table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | index | y | list[y] | list[index] | list[x] |
| 0 | 0 | 1 | 10 | 2 | 2 |
| 0 | 0 | 2 | 5 | 2 | 2 |
| 1 | 1 | 2 | 5 | 10 | 10 |
| 1 | 2 | 2 | 1 | 5 | 10 |
| 1 | 2 | 2 | 1 | 10 | 5 |

|  |  |  |
| --- | --- | --- |
| 2 | 5 | 10 |

Output:

**Bubble Sort**

**Description**

In this algorithm first largest element picked from array and placed at last place then second smallest and so on for total array. So whole array will be sorted.

**Pseudo Code**

def bubbleSort(A):

for i=1 to Length.A:

for j=1 to Length.A:

if A[j] > A[j+1]:

temp = A[j+1]

A[j+1] = A[j]

A[j] = temp

Return A

**Python Code**

    def getSortedArray(self , Array , column , asc):

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

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

                if (Array[j].data[column] > Array[j+1].data[column]) == asc:

                    temp = Array[j+1]

                    Array[j+1] = Array[j]

                    Array[j] = temp

        return Array

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Best Case **O(n)** | Worst Case **O(n^2)** | Average Case **O(n^2)** |
| When array is already sorted only for loop will execute of length n and while loop no run so only n times operation executes. | When given array is in reverse sorted order outer for loop will execute n times and inner loop will execute n time for one value of n so whole operations will be O(n^2). | When given array is in unsorted not fully reversed sort time will also about O(n^2). |

**Strengths**

* It is effective for small number of inputs.
* It is simple/easy to understand.
* It takes also less time when array is already sorted.

**Weakness**

* It takes more time for large number of inputs.
* It takes more time when array is already unsorted or near sorted position.
* Bubble sort is suitable for academic level, not for industry level.

**Proof of Correctness**

**Loop Invariant**

Total elements from last are their right position according to index value.

* **Initialization**

When value of outer loop index will be one then first largest element will be at last position in subarray.

* **Maintenance**

At the value one of index of outer loop first largest element will be at last position and when value of outer loop index will be 2 then smallest element of largest will be at last minus one position in subarray.

* **Termination**

When loop will terminate at last index of array then minimum element will be first position in so from this at right position array will be sorted so whole array will be sorted.

**Dry Run**

Input:

|  |  |  |
| --- | --- | --- |
| 2 | 3 | 5 |

Trace Table:

|  |  |  |  |
| --- | --- | --- | --- |
| i | j | list[j] | list[j+1] |
| 0 | 0 | 2 | 3 |
| 0 | 1 | 3 | 5 |
| 1 | 0 | 2 | 3 |
| 1 | 1 | 3 | 5 |
| 2 | 0 | 2 | 3 |
| 2 | 1 | 3 | 5 |

|  |  |  |
| --- | --- | --- |
| 2 | 3 | 5 |

Output:

**Merge Sort**

**Description**

In merge sort divide the array into half and then call again and again merge sort to divide the array unless array contains only one element. And then merge function merge these elements in this way that array becomes sorted.

**Pseudo Code**

def merge(A , p , q , m):

left = []

right = []

for i=p to m+1

left.append(A[i])

for j=m+1 to q

right.append(A[i])

left.append(infinity)

left.append(infinity)

i=0

j=0

for k=p to q+1

if left[i] <= right[i]:

A[k] = left[i]

i=i+1

else:

A[k] = right[j]

J=j+1

def mergeSort(A , l , m):

if l<m:

mid = l+m/2

mergeSort(A , l , mid)

mergeSort(A , mid+1 , m)

merge (A , l , m , mid)

**Python Code**

    def merge(self , Array , p , q , m , col , asce):

        left = []

        right = []

        for i in range(p , m+1):

            left.append(Array[i])

        for j in range(m+1 , q+1):

            right.append(Array[j])

        left.append(sys.maxsize)

        right.append(sys.maxsize)

        j=0

        i=0

        for k in range(p,q+1):

            if (left.data[col] <= right.data[col]) == asce:

                Array[k]=left[i]

                i+=1

            else:

                Array[k]=right[j]

                j+=1

    def merge\_sort(self , A ,   l , m , col , asce):

        if l < m:

            mid = (l+m)//2

            self.merge\_sort(A , l , mid , col , asce)

            self.merge\_sort(A , mid+1 , m , col , asce)

            self.merge(A, l , m , mid , col , asce)

    def getSortedArray(self , Array , col ,asce):

        length = len(Array)

        self.merge\_sort(Array , 0 , length-1 , col , asce)

        return Array

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Best Case **O(nlgn)** | Worst Case **O(nlgn)** | Average Case **O(nlgn)** |
| Merge sort computes nlgn when what type of given array not matter (sorted or unsorted or revrseSort) | Merge sort computes nlgn when what type of given array not matter (sorted or unsorted or revrseSort) | Merge sort computes nlgn when what type of given array not matter (sorted or unsorted or revrseSort) |

**Strengths**

* It is effective for large number of inputs.
* It is applied on any large number of inputs.

**Weakness**

* It is little bit not effective for small number of inputs as compare to insertion sort

**Proof of Correctness**

**Loop Invariant**

* **Initialization**

Prior to the first iteration of the loop k will be equal to q+1 and subarrays will empty and left array and right array are smallest elements that are not copied in the array.

* **Maintenance**

When elements are copied back into the array then left array will also be the smallest elements of the array because when element in left array will be small are equal to the left array then it will copy into array and when all elements are copied then this will also be smaller than array.

* **Termination**

At termination k will be equal to q+1 and at that step array contains all elements in sorted order.

**Dry Run**

Input:

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 4 | 6 |

Trace Table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| i | j | lb | mid | ub | k |
| 0 | 0 | 0 | 1 | 3 | 0 |
| 1 | 0 | 0 | 1 | 3 | 0 |
| 2 | 0 | 0 | 1 | 3 | 1 |
| 2 | 1 | 0 | 1 | 3 | 2 |
| 2 | 2 | 0 | 1 | 3 | 3 |

This is array maintaining during operation.

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 5 | 6 |

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 5 | 6 |

**Quick Sort**

**Description**

In this a pivot is selected from array this will be anyone from middle of array or last. Elements that are small from pivot place before pivot, and that are greater than pivot, place after pivot. Pivot is on their right place in array call the function recursively unless starting index of will not equal to ending index.

**Pseudo Code**

def quicksort(A , l , h):

if l<h:

pivot = partition(A , l , h)

quicksort(A , l , pivot-1)

quicksort(A , pivot+1 , h)

def partition(A , l , h):

pivot = A[h]

i=l-1

for j=l to high:

if A[i] < pivot:

i= i+1

temp = A[i]

A[i] = A[j]

A[j] = temp

temp = A[i+1]

A[i+1] = A[h]

A[h] = temp

**Python Code**

    def quickSort(self , A , low , high , column , asc):

        if low < high:

            pivotI = self.partitionStr(A, low, high , column , asc)

            self.quickSort(A , low , pivotI-1 , column  , asc)

            self.quickSort(A , pivotI+1 , high , column , asc)

    def partitionStr(self , Array , low , high , column , asc):

        pivot = Array[high].data[column]

        i = low-1

        for j in range(low , high):

            if (Array[j].data[column] < pivot) == asc:

                i = i+1

                tempL = Array[i]

                Array[i] = Array[j]

                Array[j] = tempL

        temp = Array[i+1]

        Array[i+1] = Array[high]

        Array[high] = temp

        return i+1

    def getSortedArray(self , Array  , column , asce):

        length = len(Array)

        self.quickSort(Array , 0 , length-1 , column , asce)

        return Array

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Best Case **O(nlgn)** | Worst Case **O(n^2)** | Average Case **O(nlgn)** |
| When pivot element is from middle element in sorted array or near to middle. | When pivot element is small element from all or large element from all then one subarray will empty and will all other numbers. | Average case occur when other two case does not occur. |

**Strengths**

* Its advantage is that its average case complexity to sort an array is fast from merge sort.
* It is also suitable for large arrays.
* It is easy to understand and implement.

**Weakness**

* It is very time consuming when pivot element is in start or in end of the array.

**Proof of Correctness**

* **Basis Step**

P (1) is true because one element always is sorted.

* **Inductive Hypothesis**

Assume that P (K) is true for k < n here n is length of array.

* **Inductive Step**

Let k(left) and k(right) are subarrays of array length of this will be less than **n** by our hypothesis these arrays will be in sorted.

**Dry Run**

Input:

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 10 | 5 | 6 |

Trace Table:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| low | high | pivot | i=low-1 | j | temp | list[j] | list[high] | list[i+1] |
| 0 | 3 | 6 | -1 | 0 |  |  |  |  |
| 0 | 3 | 6 | 0 | 0 | 2 | 2 |  |  |
| 0 | 3 | 6 | 0 | 1 |  |  |  |  |
| 0 | 3 | 6 | 1 | 2 |  |  |  |  |
| 0 | 3 | 6 | 2 | 2 | 5 | 5 |  |  |
| 0 | 3 | 6 | 2 |  | 6 |  | 6 | 6 |
| 0 | 3 | 6 |  |  | 10 |  | 6 | 6 |

Then again recursive call make this array sorted.

Output:

|  |  |  |
| --- | --- | --- |
| 2 | 5 | 10 |

**Heap Sort**

**Description**

Like selection sort heap sort divide the array into sorted and unsorted region and then iteratively minimum the unsorted region step by step by extracting largest element from it and inserting it into sorted region.

**Pseudo Code**

def heap(A , n , index):

large = index

left = 2\*index+1

right = 2\*index+2

if left < n and A[large] < A[left]

large = left

if right < n and A[large] < A[right]

large = right

if large != index:

A[index], A[large] = A[large] , A[index]

Heap(A , n , large)

def heapsort(A):

n = Length.A

for i=n/2 to 1

heap(A , n , i)

for i=n-1 to 1

A[i], A[0] = A[0] , A[i]

heap(A , i , 0)

**Python Code**

    def heap(self , A , n , index , col , asce):

        large = index

        l = 2\*index+1

        r = 2\*index+2

        if l < n and ((A[large].data[col] < A[l].data[col]) == asce):

            large = l

        if r < n and ((A[large].data[col] < A[r].data[col]) == asce):

            large = r

        if large != index:

            A[index], A[large] = A[large] , A[index]

            self.heap(A , n , large , col , asce)

    def heapSort(self , A , col , asce):

        n = len(A)

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

            self.heap(A , n , i , col , asce)

        for i in range(n-1 , 0 , -1):

            A[i] , A[0] = A[0], A[i]

            self.heap(A , i , 0 , col , asce)

    def getSortedArray(self , Array , col , asce):

        self.heapSort(Array , col , asce)

        return Array

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Best Case **O(nlgn)** | Worst Case **O(nlgn)** | Average Case **O(nlgn)** |
|  |  |  |
| Heap sort takes time nlgn because height of a tree is nlgn any comparison base algorithm cannot sort algorithm less than operations nlgn | Heap sort takes time nlgn because height of a tree is nlgn any comparison base algorithm cannot sort algorithm less than operations nlgn | Heap sort takes time nlgn because height of a tree is nlgn any comparison base algorithm cannot sort algorithm less than operations nlgn |

**Strengths**

* It is an in-place sorting algorithm.
* It is widely used for its efficiency.

**Weakness**

* It makes tree of sorting algorithms.

**Proof of Correctness**

**Loop Invariant**

We need to prove build maxheap used for sorting maintain invariant throughout the algorithm.

* **Initialization**

Before first iteration of the loop. Everything is a leaf and it is sorted.

* **Maintenance**

Assume we have working solution till now. Children of node I are maximum than I. So, this preserves the loop invariant.

* **Termination**

When loops terminate, I will be zero and from loop invariant each node is root of maximum heap.

**Dry Run**

Input:

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

Call maximum heap

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

For i=3

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

For i=2:

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

For i=1:

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

For i=0:

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

**Shell Sort**

**Description**

Shell do not take large shifts as in case of insertion sort. If small value is very right and has to moved far left. This algorithm uses insertion sort on widely spread elements, first to sort them and then sorts less widely spaced elements. Spacing is termed as interval.

**Pseudo Code**

def shellSort(A)

interval = Length.A

while interval > 0

i=0

j=0

while j < len(A)

if A[i] > A[j]

swap A[i] and A[j]

i=i+1

j=j+1

k=1

while k-interval > -1

if A[k-interval] > A[k}

swap A[k-space] and A[k]

k = k-1

interval= interval/2

**Python Code**

    def getSortedArray(self , Array , col ,  asce):

        space = len(Array)

        while space > 0:

            i = 0

            j = space

            while j < len(Array):

                if (Array[i].data[col] > Array[j].data[col]) == asce:

                    temp = Array[i]

                    Array[i] = Array[j]

                    Array[j] = temp

                i = i+1

                j = j+1

                k = 1

                while k -space > -1:

                    if (Array[k-space].data[col] > Array[k].data[col]) == asce:

                        temp = Array[k-space]

                        Array[k-space] = Array[k]

                        Array[k] = temp

                    k = k-1

            space = space//2

        return Array

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Best Case **O(nlgn)** | Worst Case **O(n^2)** | Average Case **O(nlgn)** |
| When array is already in sorted order shell sort takes times nlgn on operations. | When array is reversed sorted order or near to this then shell sort takes times on operations less than or equal to n^2 | It occurs when array is in between sorted and unsorted order. |

**Strengths**

* Shell sort is 5.32times faster than bubble sort.
* Shell sort takes less time when array is already in sorted order.

**Weakness**

* It is an unstable sorting algorithm.
* It is difficult to understand algorithm.
* It takes more time when array in reverse sorted order.

**Proof of Correctness**

**Loop Invariant**

At every step index I all values in subarray are sorted.

* **Initialization**

Array start with the first element in the subarray will be in sorted order.

* **Maintenance**

When loop will execute on index, I then array at position I will be sorted and at the next iteration of the loop at the index I array will obviously sorted at position I Left side of I whole array will be sorted.

* **Termination**

Loop will terminate when I will be on last index of the array and at that time whole array will sorted.

**Dry Run**

Input:

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

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

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 1 | 3 | 4 |

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

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

**Counting Sort**

**Description**

Counting sort is a stable sorting algorithm, which is used to sort according the keys that are in array indexes. It counts the number of values that are same.

**Pseudo Code**

def findLargest(A):

large = -infinity

for i=1 to Length.A

if A[i] > large:

large = A[i]

return large

def findsmallest(A):

small = infinity

for i=1 to Length.A

if A[i] < large:

small = A[i]

return small

def countingSort(A):

small = findSmallest(A)

if small < 0:

adder = small\*-1

for i=1 to Length.A

A[i] = A[i] +adder

    Count = []

B = []

K = findLargest(A)

for i=1 to Length.A

B.append(0)

for i=1 to k+1

count.append(0)

for j=1 to Length.A

count[A[j]] = count[A[j]]+1

for i=2 to k+1

count[j] = count[j]+ count[j-1]

for i= Length.A downto 1

count[A[i]] = count[A[i]] -1

B[count[Ai]] = A[i]

If small < 0:

for i=1 to Length.A

B[i] = B[i] +small

**Python Code**

    def findLargest(self , Aray , col):

        largest = -100000

        for i in range(0 , len(Aray)):

            if Aray[i].data[col] > largest:

                largest = Aray[i].data[col]

        return largest

    def findSmallest(self , Aray , col):

        smallest = 100000

        for i in range(0 , len(Aray)):

            if Aray[i].data[col] < smallest:

                smallest = Aray[i].data[col]

        return smallest

    def getSortedArray(self , Array , col ,  asce):

        if type(Array[3].data[col]) == int:

            smaller = self.findSmallest(Array , col)

            if smaller < 0:

                adder = smaller\*-1

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

                    Array[i].data[col] = Array[i].data[col]+adder

            count = []

            B = []

            k = self.findLargest(Array , col)

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

                B.append(0)

            for i in range(0 , k+1): # Array of zeros of k+1 length

                count.append(0)

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

                count[Array[j].data[col]] = count[Array[j].data[col]] + 1

            for j in range(1 , k+1):

                count[j] = count[j]+count[j-1]

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

                count[Array[i].data[col]] = count[Array[i].data[col]]- 1

                B[count[Array[i].data[col]]] = Array[i]

            if smaller < 0:

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

                    B[i].data[col] = B[i].data[col]+smaller

            if asce==True:

                return B

        else:

            count = []

            B = []

            for i in range(0 , 256):

                B.append(0)

            for i in range(0 , 256): # Array of zeros

                count.append(0)

            answer = ["" for \_ in Array]

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

                sAttribute = Array[j].data[col]

                index = ord(sAttribute[0])

                count[index] = count[index] + 1

            for j in range(1 , 256):

                count[j] = count[j]+count[j-1]

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

                sAttribute = Array[i].data[col]

                index = ord(sAttribute[0])

                B[count[index]] = Array[i]

                count[index] = count[index]- 1

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

                answer.append(B[i])

            return answer

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Best Case **O(n)** | Worst Case **O(n)** | Average Case **O(n)** |
| All loop executes n times no matter how type of array is sorted or unsorted so time complexity is 4n in terms of big o notation O(n) | All loop executes n times no matter how type of array is sorted or unsorted so time complexity is 4n in terms of big o notation O(n) | All loop executes n times no matter how type of array is sorted or unsorted so time complexity is 4n in terms of big o notation O(n) |

**Strengths**

* It is stable sort.
* It sort items in liner time.
* It is fast as compare to other algorithms.

**Weakness**

* It takes more space when array contain elements of large numbers.
* It is not suitable for sorting string values.

**Proof of Correctness**

* **Basis Step**

P (1) is true because one element always is sorted.

* **Inductive Step**

Assume we can sort k-1 numbers. Consider sorting numbers of k digits. First sort first k-1 digits and then last k-1 sorts.

* **Inductive Step**
* Kth sorts last digit correctly. Assume n1 and n2 are two digits and suppose n1<n2 then n2 will proceed n1. If n1 and n2 are same then they will be same by sorting algorithms and if according to kth digits and so k-1 and k-1 will be sorted.

**Dry Run**

Input:

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

outPutArray:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |

count:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |

count:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 0 | 1 |

count:

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

count:

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

outPutArray:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |

#When loop first time iterate from high index to low

count:

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

outPutArray:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 0 | 0 |

#When loop second time iterate from high index to low

count:

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

outPutArray:

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

#When loop third time iterate from high index to low

count:

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

outPutArray:

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

#When loop fourth time iterate from high index to low

Output:

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

**Bucket Sort**

**Description**

In bucket sort elements should be in range of zero and one. Make buckets of these number so size of array will be small. And then apply insertion because we know that for small number of elements insertion sort is better.

**Pseudo Code**

def bucketSort(A)

bucket #make 2d list of length of given array

for i=1 to Length.A

index = 10\*A[i]

bucket[index].append(A[i])

for i=1 to Length.A

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

C = []

i=0

for k in bucket

for j=1 to Length.A

C.append(k[j]

**Python Code**

    def insertionSort(Array):

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

            key = Array[i]

            j = i-1

            while j>=0 and key<Array[j]:

                Array[j+1] = Array[j]

                j = j-1

            Array[j+1] = key

        return Array

    def getSortedArray(self , Array , prdType , asc):

        if type(prdType) == int:

            bucket = [[]  for i in range(len(Array))]

            for i in range(len(Array)):

                index = int(10\*Array[i])

                bucket[index].append(Array[i])

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

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

            C = []

            i = 0

            if asc == False:

                for k in reversed(bucket):

                    for j in reversed( range(0 , len(k))):

                        C.append(k[j])

                return C

            else:

                for k in bucket:

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

                        C.append(k[j])

                return C

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Best Case **O(n)** | Worst Case **O(n^2)** | Average Case **O(n)** |
| If elements in the buckets are uniformly distributed and also times complexity becomes better when elements in bucket are in ascending order. | If elements in buckets are much more and in reverse sorted order than time complexity will depend on algorithm which is used for sorting so operation takes times O(n^2) | If elements are randomly distributed in the array then it will take linear time. |

**Strengths**

* It is linear time sorting algorithm.
* It is fast for large arrays also as compared to quicksort.
* It is a stable sort algorithm

**Weakness**

* Bucket sort is not useful when buckets made very large.
* It is not for non-stable implementation

**Proof of Correctness**

* **Basis Step**

P (1) is true because one element always is sorted.

* **Inductive Step**

Assume we can sort k-1 numbers. Consider sorting numbers of k digits. First sort first k-1 digits and then last k-1 sorts.

* **Inductive Step**

Kth sorts last digit correctly. Assume n1 and n2 are two digits and suppose n1<n2 then n2 will proceed n1. If n1 and n2 are same then they will be same by sorting algorithms and if according to kth digits and so k-1 and k-1 will be sorted.

**Dry Run**

Input:

|  |  |  |  |
| --- | --- | --- | --- |
| 0.2 | 0.1 | 0.4 | 0.2 |

bucket = [[]]

Trace Table:

|  |  |
| --- | --- |
| i | bucket[i] |
| 0 | 0.1 Apply insertionSort |
| 1 | [0.2][0.2] apply insertion Sort |
| 2 | 0.4 Apply insertion Sort |
| 3 | Empty[] |

C = []

|  |  |  |
| --- | --- | --- |
| k | bucket[k] | C |
| 0 | 0.1 | 0.1 |
| 1 | [0.2] [0.2] | [0.1, 0.2, 0.2] |
| 2 | 0.4 | [0.1, 0.2, 0.2, 0.4] |

**Search**

**Linear Search**

**Description**

Linear Search is a searching algorithm to search data from array. It searches data from array in such a way that it starts comparing the data from the first index of the loop to the last index of loop returns that data.

**Pseudo Code**

def search(toSearch , Array):

searchFind = 0

for i=1 to Array.Lenght

if Array[i] == toSearch

temp = Array[searchFind]

Array[searchFind] = Array[i]

Array[i] = temp

searchFind = searchFind+1

return Array

**Python Code**

    def search(self , toSearch, Array , column):

        searchFindInArray = 0

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

            if Array[i].data[column] == toSearch:

                temp = Array[searchFindInArray]

                Array[searchFindInArray] = Array[i]

                Array[i] = temp

                searchFindInArray = searchFindInArray+1

        return Array

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Best Case **O(n)** | Worst Case **O(n)** | Average Case **O(n)** |
| Linear search time complexity is O(n) no matter array is in reverse order or in sorted order. Because it compares search with index of the array from first to last. | Linear search time complexity is O(n) no matter array is in reverse order or in sorted order. Because it compares search with index of the array from first to last. | Linear search time complexity is O(n) no matter array is in reverse order or in sorted order. Because it compares search with index of the array from first to last. |

**Strengths**

* It is very simple.
* It is very easy to implement.
* It is very efficient.

**Weakness**

* It takes more time to for very huge amount of data.
* Its time complexity is O(n) is more form many other searching algorithms.

**Proof of Correctness**

**Loop Invariant**

At every step index I all values will be in start of the array which are equal to search text.

* **Initialization**

At the start of the loop searchFind in array has zero value.

* **Maintenance**

When loop will execute on index the values of searchFind will be at that time the search meet with array index.

* **Termination**

Loop will terminate when index will be the equal to the length of the array. At that time also searchFind value will be how much time search meet in array.

**Dry Run**

Input:

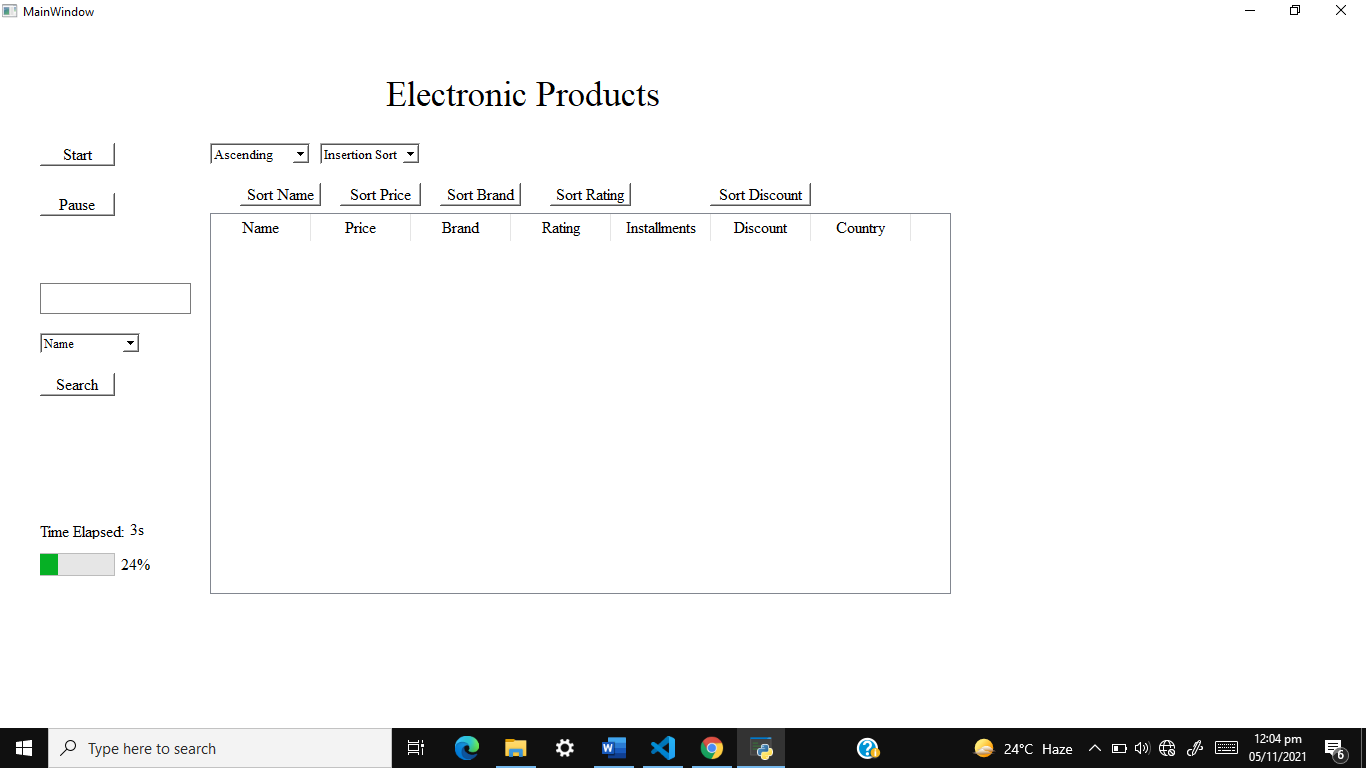
|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 4 | 6 |

Search = 4

Trace Table:

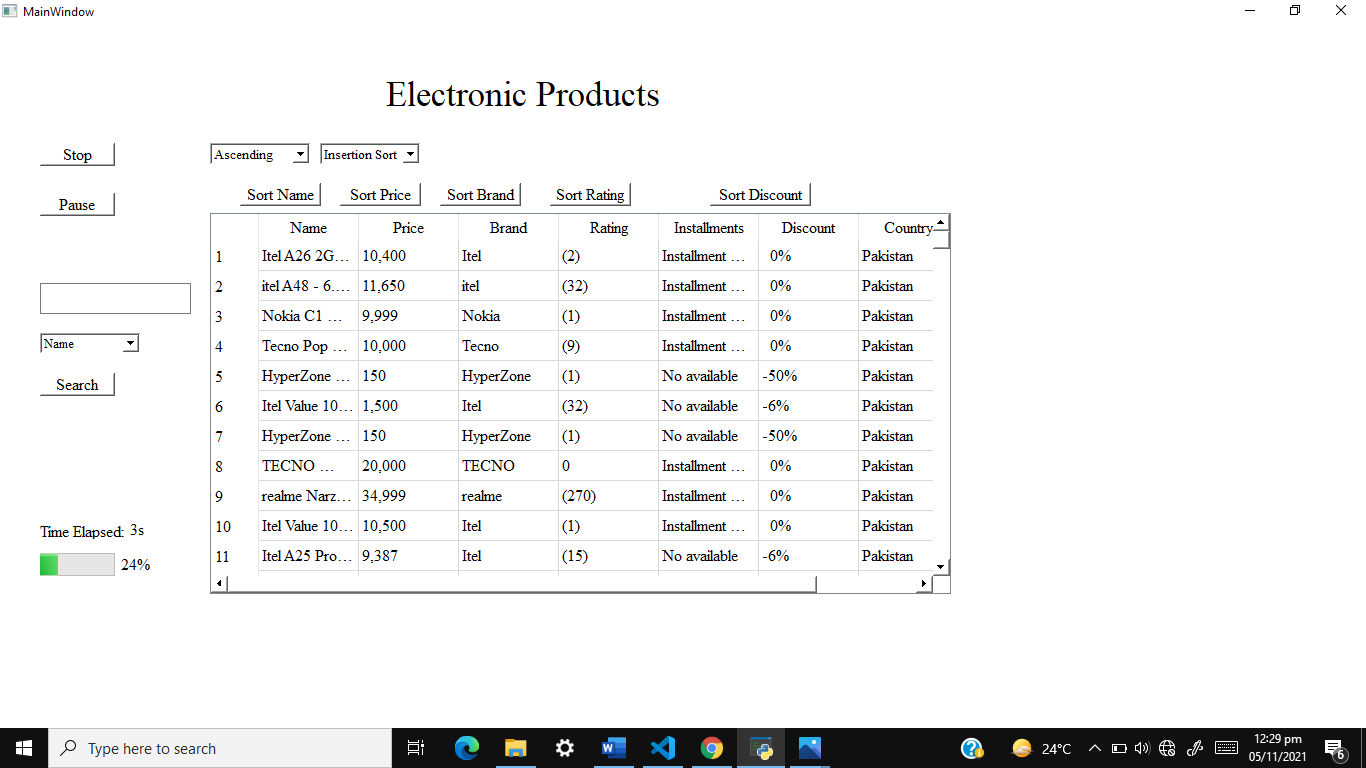
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| i | search | Array[i] | searchFind | Array[i] this will be return |
| 0 | 4 | 2 | 0 | 2,5,4,6 |
| 1 | 4 | 5 | 0 | 2,5,4,6 |
| 2 | 4 | 4 | 0 | 4,5,2,6 |
| 3 | 4 | 6 | 1 | 4,2,5,6 |

**User Interface**

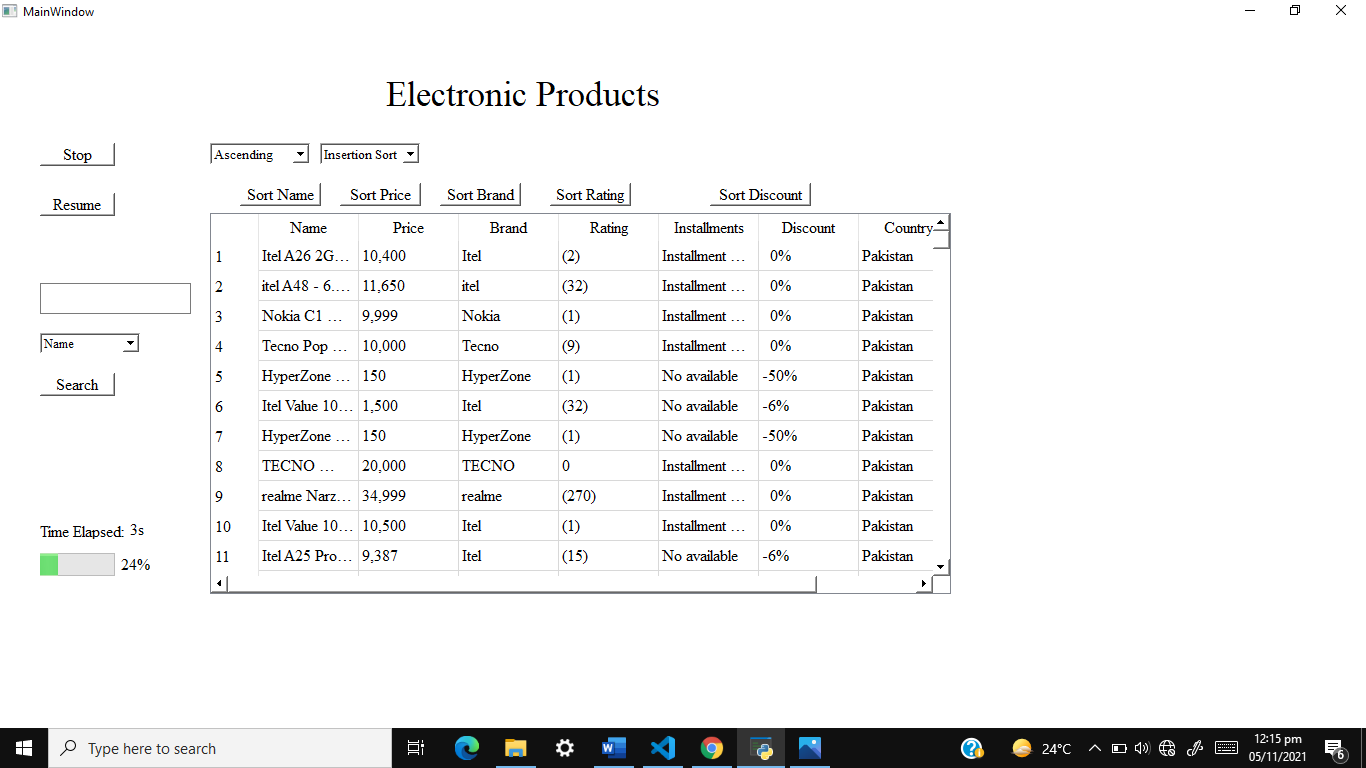


**Integration**

In integration I merge sorting and searching algorithms with graphical user interface which was design in pyqt5 designer. There are many problems which I face during integration. In am only one member of my group. All other students made their groups of two members. I was one student which cannot make their group because there is no any student in the class from which I make group. So, I start project lonely. I write all sorting problems lonely. I use many times to do this and face many difficulties. Because there is difficult to do all things for a single person. And in integration I face also many difficulties and complete work with more difficult. When I integrate our algorithms with User interface code then face many errors in the code. And then I solve these errors by using days and nights. There is more difficult for me because I was a single person. But I solve these errors and now I am very happy for this. In integration I face an error of table Widget. When I send data to table Widget then it will give me error of attribute that table Widget is not defined. Then this error takes me a whole day to solve. Then other errors take little or more time to solve. I take little bit help from my class fellows to solve the errors and about project information. I take help form internet to solve errors from different websites.



**Figure1**



**Figure2**

**User Interface Explanation**

In **Figure1** **Start Button’s** function is to start scraping of data and program. When we click on start button text and functionality of button will change and it will be converted into to **Stop Button as shown in Figure2.** Stop Button’s function is to stop scraping.

In **Figure1** **Pause Button’s** function is to pause scraping of data. When we click on Pause button text and functionality of button will change and it will be converted into to **Resume Button as shown in Figure2. A**nd again, by clicking on Resume button scraping will resume and text and functionality of button will be converted into pause button. This procedure will be continued as program continued.

In **Figure**s Label is for to take input from user to search data according to any attribute. Below there is **Drop Down** which will allow users to select that attribute which he/she want to search. Below there is **Search** Button by clicking on this search will start.

There are seven columns in table as shown in **Figure.** First column will contain name of product, second will contain price, third will contain brand name, fourth will contain ratings, fifth will contain installments, six will contain discounts (Discount will be percentages) and seven will contain country name.

In **Figures** on above there is Drop down for ascending and descending. This will allow users to select that they want to sort in data in ascending or descending order.

And on the right side of ascending and descending drop down there is a Drop down for sorting algorithms. This will allow users to select sorting algorithm and apply on sorting which he/she want.

On above columns of Name, Price, Brand, Discount and Rating. Buttons for sorting these attributes. User will click on which button data will be sort according to that attribute.

**Collaboration**

There is no collaboration from my team member. Because I was only one member in my group. I handle all errors problems, write code, write documents lonely.

**Classes and Functions**

**Classes**

**Ui\_TimeUse**

This class will contain all code of graphical user interface.

**Electronics**

This class will contain an attribute of data whose data type is List. And a constructor which will take attributes to scrap.

**Insertion Sort**

This class will contain function for insertion sort.

**Bubble Sort**

This class will contain function for bubble sort.

**Selection Sort**

This class will contain function for selection sort.

**Merge Sort**

This class will contain function for merge sort.

**Quick Sort**

This class will contain function for quick sort.

**Heap Sort**

This class will contain function for heap sort.

**Shell Sort**

This class will contain function for shell sort.

**Counting Sort**

This class will contain function for counting sort.

**Bucket Sort**

This class will contain function for bucket sort.

**Radix Sort**

This class will contain function for radix sort.

**Functions**

**Laptops Scrap**

This scrap all data of laptop from daraz.pk.

**Watch Scrap**

This scrap all data of watches from daraz.pk.

**Mobile Phones Scrap**

This scrap all data of mobile phones from daraz.pk.

**Mobile Accessories Scrap**

This scrap all data of mobile accessories from daraz.pk.

**LED Scrap**

This scrap all data of Leds from daraz.pk.

**Televisions Scrap**

This scrap all data of televisions from daraz.pk.

**Cameras Scrap**

This scrap all data of cameras from daraz.pk.

**Desktop Components Scrap**

This scrap all data of desktop components from daraz.pk.

**Refrigerator Scrap**

This scrap all data of Refrigerator from daraz.pk.

**Oven Scrap**

This scrap all data of ovens from daraz.pk.

**Air Conditioner Scrap**

This scrap all data of air conditioners from daraz.pk.

**Air Cooler Scrap**

This scrap all data of air cooler from daraz.pk.

**UPS Scarp**

This scrap all data of ups from daraz.pk.

**Generators Scrap**

This scrap all data of generators from daraz.pk.

**Washer Dryers Scrap**

This scrap all data of washer dryers from daraz.pk.

**Water Dispensers Scrap**

This scrap all data of water dispensers from daraz.pk.

**Vacuum Cleaner Scrap**

This scrap all data of vacuum cleaner from daraz.pk.

**Fans Scrap**

This scrap all data of fans from daraz.pk.

**Irons Scrap**

This scrap all data of irons from daraz.pk.