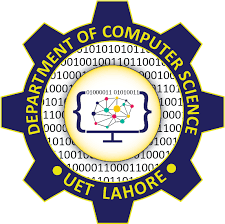
**Final Report for Mid Term Project**

**Warehouse management system**



**Submitted To:**

Mr. Samyan Qayyum Wahla

**Submitted By:**

Shakeel Ahmed 2021-CS-03

M Uzair 2021-CS-17

AmadIrfan 2021-CS-25

**Project ID: G44**

University of Engineering and Technology

Lahore, Pakistan

**Table of Contents:**

1. Learning Outcomes
2. Executive Summary
3. Scrapping

3.1 Problem Statement

3.2 Project Overview

3.3 Attributes

3.4 Algorithms

3.4.1 for Sorting

3.4.2 for Searching

3.5 Obstacles

3.6 Resources for Development

4- Description for Algorithms

4.1 Algorithm Explanation

4.2 Pseudo Code

4.3 Time Complexity

4.4 Strength

4.5 Weakness

4.6 Dry run

1. GUI of the Software

5.1 Login Page UI

5.2 Profile Page UI

5.3 Products Page UI

5.4 Change Product Info

5.5 Scrap Data

**Learning Outcomes:**

* In this project we learnt scrapping data from websites, using Selenium.
* We learnt to store data in csv files using pandas in python
* We learnt how to design UI and connect it with the backend to using python.
* We learnt implementing multiple functions on data like, sorting, Multiple level Sorting, searching.

**Executive Summary:**

The main page will be reached by him. view product, and scrap product are the five possibilities.

The user can view all of the available products in the view product section. The products can be sorted by the user based on his needs. The user may choose any of the ten sorting algorithms that are provided.

The user can choose any one of the seven product qualities and sort by it

**Scrapping:**

https://www.daraz.pk/

We scrapped data from the upper link.

1. **Problem Statement:**

Anyone starting a new business nowadays needs to be knowledgeable about every product he plans to offer. He needs a resource where he can access all the necessary product information

1. **Project Overview:**

This software was created to address the issue raised in Point 1. The user will benefit from this software's assistance in obtaining product information. The customer will attempt to purchase all of the products with high ratings because the software can obtain the ratings of the scraped products, giving him a sense of which product consumers prefer the best. The user will be able to earn more money as a result. The user can choose the things for his store from the product's scrapping price that he can pay.

As soon as a new product is introduced to the market, it is updated on this page. In order to see the newly released products

**Attributes:**

* id
* Brand
* Price
* Discount
* Seller
* Instalment
* Return policy

**Algorithms:**

1. **For Sorting:**

* Merge Sort
* Insertion Sort
* Shell Sort
* Heap Sort
* Selection Sort
* Bubble Sort
* Quick Sort
* Counting Sort

1. **For Searching:**

* Linear Search
* Binary search

1. **Obstacles:**

Some of the websites does not allow us to scrape their data. While the website we are using enables us to scrap data up to five pages. But we used a link in which the structure of the page is changed and the new products are added in the same pages. And when working on large data for example on lac products the software hangs, sorting takes a lot of time.

1. **Resources for development:**

* Spyder as IDE
* Visual Studio code
* Python as Programming Language
* PYQT5 for the Designing of UI

**Description for Algorithms:**

**Merge Sort:**

Merge sort is a comparison based algorithm. It works on the divide and conquer principle. It divides the array into small parts until there are only two elements left. Then it swaps the elements and sorts them. When all the elements are sorted it again merges them to an array.

**Pseudocode:**

**Merge-Sort(array, p , r, col-index)**

if p != r then

Let q = ((p+r)/2) (Round and return the largest closest number)

Merge-Sort(array, p, q, col-index)

Merge-Sort(array, q+1, r ,col-index)

Merge(array, p, q, r ,col-index)

return array

**Merge(array, p, q, r , method)**

Let n1 = q-p

Let n2 = r-q

Let Left-array = []

Let Right-array = []

for i = 0 to n1

append array[i+p] in Left-array

for i = 0 to n2

append array[i+q] in Right-array

Let i = 0

Let j = 0

Let k = p

Call Selection-Sort (left-array, col-index)

Call Selection-Sort (right-array, col-index)

while i < left-array.length and j<right-array.length

if left-array[i][col-index] < right-array[j][col-index]

append the elements from the left array to array[k]

else

append the elements from the left array to array[k]

k +=1

while i < left-array.length

append remaining elements of left-array to array[k]

while j < len(right\_arry)

append remaining elements of right-array to array[k]

**Python Code**:

**def MergeSort(self,arry,p ,r, method):**

if p != r :

q = int((p+r)/2)

self.MergeSort(arry,p,q,col\_index)

self.MergeSort(arry, q+1, r,col\_index)

self.Merge(arry,p,q,r,col\_index)

return arry

**def Merge(self,arry,p,q,r ,method ):**

n1 = int(q-p)

n2 = int(r-q)

left\_arry = []

right\_arry = []

for i in range(0,n1):

left\_arry.append(arry[i+p])

for i in range(0,n2):

right\_arry.append(arry[i+q])

i = 0

j = 0

k = p

self.selection(left\_arry,col\_index)

self.selection(right\_arry,col\_index)

while i<len(left\_arry) and j<len(right\_arry) :

if left\_arry[i][col\_index]<right\_arry[j][col\_index]:

arry[k] = left\_arry[i]

i +=1

else:

arry[k] = right\_arry[j]

j +=1

k +=1

while i < len(left\_arry):

arry[k] = left\_arry[i]

i += 1

k += 1

while j < len(right\_arry):

arry[k] = right\_arry[j]

j += 1

k += 1

**Time Complexity:**

Worst Complexity: O(n\*log(n))

Average Complexity: O(n\*log(n))

Best Complexity: O(n\*log(n))

**Strength:**

It is faster while working with the large amount of data.

**Weakness:**

It is slower while working for the small amount of data.

**Dry Run:**

**Split Merge**

**Insertion Sort:**

Insertion sort is a comparison based algorithm in this algorithm array is divided into two parts sorted elements are put on one side and the unsorted on the other side. The algorithm keeps finding the elements from the unsort part and put it in the sorted part of the array.

**Pseudocode:**

insertionSort(A,method)

for i = 1 to A.length

key=A[i]

for j=i-1 to A.length

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

swap(A[j], A[i])

**Python Code**:

def Insertion\_Sort(self, method) :

for i in range(1,len(self.arry)):

Arry\_Inside\_As\_key = self.arry[i]

j = i -1

for j in range(len(self.arry)):

Arry\_Inside = self.arry[j]

if (Arry\_Inside\_As\_key)< (self.arry[j]):

temp = self.arry[j]

self.arry[j] = self.arry[i]

self.arry[i] = temp

**Time Complexity:**

Worst Complexity: O(n2)

Average Complexity: O(n2)

Best Complexity: O(n)

**Strength:**

It works faster for the small amount of data.

**Weakness:**

It works slower if the amount of data is large.

**Dry Run:**

**Shell Sort:**

Shell sort is a comparison based algorithm. It works similar to insertion sort. But the difference is that it decides the gap between the number first and swaps the elements with same interval first and then the others.

**Pseudocode:**

**Shell-Sort(A, n, column-index) #n is the length of array**

h=n/2

while (h>0)

for i = h to n

t=A[i]

j=i

while (j>=h and (A[j-h] [column-index])>A[i] [column-index])

Swap(A[j], A[j-h]

j -= h

A[j] = t

**Python Code:**

def shell\_sort(self,inp, n, col\_index):

h = n // 2

while h > 0:

for i in range(h, n):

t = inp[i]

j = i

while j >= h and (inp[j - h][col\_index]) > (inp[i][col\_index]):

inp[j] = inp[j - h]

j -= h

inp[j] = t

h = h // 2

**Time Complexity:**

Worst Complexity: O(n2)

Average Complexity: O(n\*log(n))

Best Complexity: O(1)

**Strength:**

Shell sort is 5.32 x faster than the bubble sort.

**Weakness:**

The structure of the shell sort is very complex.

**Heap Sort:**

Heap sort is a comparison algorithm. It works on max-heapify or min-heapify. For example, in the max-heapify it finds the largest element and take it the place of the root element and the root element comes at the position of the child element. In this way a tree is created and the array is sorted.

**Pseudocode:**

Heapify(array, n, i , col-index )

let maximum = i

let left = 2\*i +1

let right = 2\*i + 2

if left <n and (array[i][col-index]) <(array[left][col-index]) then

maximum = left

if Right <n and (array[maximum][col-index]) < (array[Right][col-index]) then

maximum = Right

if maximum (not equal to) i then

swap(array[i], array[maximum]

Heapify (array, n, maximum, col-index)

Heap-Sort(array, col-index) :

Let n = array.length

for i = n/2(in integer) down to 0

Heapify(array, n, i, col-index)

for I = n-1 down to 0

swap(array[0], array[i])

Heapify (array, i , 0, col-index)

**Python Code:**

def Heapify(self,array, n, i ,col\_index ):

maximum = i

left\_Child = 2\*i +1

Right\_Child = 2\*i + 2

if left\_Child <n and (array[i][col\_index]) <(array[left\_Child][col\_index]):

maximum = left\_Child

if Right\_Child <n and(array[maximum][col\_index])<(array[Right\_Child][col\_index]):

maximum = Right\_Child

if maximum != i :

temp = array[i]

array[i] = array[maximum]

array[maximum] = temp

self.Heapify(array, n, maximum,col\_index)

def Heapsort(self,arry,col\_index) :

n = len(arry)

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

self.Heapify(arry, n ,i,col\_index)

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

temp = arry[0]

arry[0] = arry[i]

arry[i] = temp

self.Heapify(arry, i, 0,col\_index)

**Time Complexity:**

Worst Complexity: O(n\*log(n))

Average Complexity: O(n\*log(n))

Best Complexity: O(n\*log(n))

**Strength:**

Memory can be allocated and deallocated any time, therefor it is considered very flexible.

**Weakness:**

Slower for small amount of data as it has to find max element every time.

**Dry Run:**

**Selection Sort:**

Selection sort is a comparison based algorithm it finds the smallest number and then takes it to the first index of the array then finds the second smallest number and put it to the second index and similarly for others. In the last array is sorted.

**Pseudocode:**

**Selection-Sort(array, col-index):**

for i = 1 to array.length

let min-index = i

for j= i+1 to array.length

let array-Inside = array[j]

let array-Inside-At-Min-index = array[min-index]

if col-index != 0 then

if array-Inside[col-index]) < (array-Inside-At-Min-index[col-index] then

min-index = j

array-Inside-At-Min-index = array[j]

Swap (array[min-index], array[i])

**Python Code:**

def selection(self,arry,col\_index):

#complete selection sort

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

min\_index = i

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

Arry\_Inside = arry[j]

Arry\_Inside\_At\_Min\_index = arry[min\_index]

if col\_index != 0:

if (Arry\_Inside[col\_index]) < (Arry\_Inside\_At\_Min\_index[col\_index)

min\_index = j

Arry\_Inside\_At\_Min\_index = arry[j]

temp = arry[min\_index]

arry[min\_index] = arry[i]

arry[i] = temp

**Time Complexity:**

Worst Complexity: O(n2)

Average Complexity: O(n2)

Best Complexity: O(n2)

**Strength:**

No additional temporary memory storage is need for the selection sort. It sorts the elements in the same array.

**Weakness:**

With the increase in the amount of the data it becomes slower.

**Dry Run:**

**Pigeonhole Sort:**

Pigeonhole is a linear sorting algorithm. It is a sorting algorithm in which number of list elements and keys are almost same.

**Pseudocode:**

**Pigeonhole-sorting(array, col-index)**

Let col-Price = [val[col-index] for val in array]

Let Max-Element = (Get maximum price of col-Price)

Let Min-Element = (Get minimum price of col-Price)

Let rang = Max-Element – Min-Element + 1

Count = array of 0s of size rang

Output = array of 0s of size array.length

for i= 0 to array.length

Count[array[i][col-index]-Min-Element] +=1

for i =1 to Count.length

Count[i] += Count[i-1]

for i=array.length down to -1

j = Count[array[i][col-index]-Min-Element] - 1

Output[j] = array[i]

Count[array[i][col-index]-Min-Element] -=1

return (Output)

**Python Code:**

**def Pigeonhole\_sorting(self,Arry, col\_index) :**

col\_Price = [val[self.col\_index] for val in Arry]

Max\_Element = max(col\_Price)

Min\_Elemenet = min(col\_Price)

rang = Max\_Element - Min\_Elemenet + 1

Count = [0]\*(rang)

Output = [0]\*len(Arry)

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

Count[Arry[i][col\_index]-Min\_Elemenet] +=1

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

Count[i] += Count[i-1]

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

j = Count[Arry[i][col\_index]-Min\_Elemenet] - 1

Output[j] = Arry[i]

Count[Arry[i][col\_index]-Min\_Elemenet] -=1

return (Output)

**Time Complexity:**

Worst Complexity: O(n+2k)

Average Complexity: O(n+2k)

Best Complexity: O(2k)

**Strength:**

Since pigeonhole sort does not need to compare the elements, therefore its working is very fast.

**Weakness:**

In pigeon knowing the range of the elements is very difficult.

**Dry Run:**

**Bubble Sort:**

Bubble sort is a comparison based algorithm It compares two adjacent elements and swap them then same for the next two elements. It does this process until all the elements are sorted in the array.

**Pseudocode:**

**Bubble-Sort(col-index)**

for i=0 to array.length

for j = 1 to array.length

let array-Inside = array[j]

let array-Inside-Next = array[j+1]

If col-index != 0 then

if (array-Inside[col-index]) > (array-Inside-Next[col-index]):

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

**Python Code:**

**def bubble(self,col\_index) :**

#complete bubble Sort

for i in range(len(self.arry )):

for j in range(1,len(self.arry )-1):

Arry\_Inside = self.arry[j]

Arry\_Inside\_Next = self.arry[j+1]

if col\_index != 0 :

if (Arry\_Inside[col\_index])>(Arry\_Inside\_Next[col\_index]):

temp = self.arry[j]

self.arry[j] = self.arry[j+1]

self.arry[j+1] = temp

**Time Complexity:**

Worst Complexity: O(n2)

Average Complexity: O(n2)

Best Complexity: O(n)

**Strength:**

Its code is very small and easy to understand. It is faster for the small amount of data and for the data that is mostly in order.

**Weakness:**

It takes a lot of time for sorting large amount of data. It can take O(n2) time.

**Dry Run:**

**Quick Sort:**

It is a comparison based sorting algorithm. It has a pivot point. In this algorithm the array is broken into small parts then it makes comparison of the elements around the pivot and arranges the elements. In the last the array is again combined. The resulted array is sorted now.

**Pseudocode:**

**Quick-Sort(array, Start, End , col-index):**

if Start<End then

Let Pivot = partition(array, Start, End, col-index)

Call Quick-Sort(array, Start, Pivot-1, col-index)

Call Quick-Sort(array, Pivot+1, End, col-index)

**partition(array, Start, End, col-index):**

Let Pivot = array[End][col-index]

p = Start - 1

for i = Start to End

if col-index != 0 then

if Pivot >= (array[i][col-index]) then

p+=1

swap(array[p], array[i])

swap(array[p+1], array[End]

return(p+1)

**Python Code:**

**def Quick\_Sort(self,Arry,Start,End , col\_index):**

if Start<End:

location\_Pivoit = self.partision(Arry,Start,End,col\_index)

self.Quick\_Sort(Arry,Start,location\_Pivoit-1,col\_index)

self.Quick\_Sort(Arry,location\_Pivoit+1 ,End,col\_index)

**def partision(self,Arry,Start,End,col\_index):**

Pivoit = Arry[End][col\_index]

p = Start - 1

for i in range(Start,End):

if col\_index != 0:

if (Pivoit)>= (Arry[i][col\_index]):

p+=1

temp = Arry[i]

Arry[i] = Arry[p]

Arry[p] = temp

temp = Arry[End]

Arry[End] = Arry[p+1]

Arry[p+1] = temp

return(p+1)

**Time Complexity:**

Worst Complexity: O(n2)

Average Complexity: O(n\*log(n))

Best Complexity:O(n\*log(n))

**Strength:**

Quick Sort is very fast in sorting the elements. As it has to decide from the pivot point to sort the elements.

**Weakness:**

Quick sort is recursive there for it can take O(n2) time to sort the elements in the worst case.

**Counting Sort:**

Counting Sort is a linear algorithm that sorts the elements of and array by counting the number of occurrences of each unique element in the array. If searches for the occurrences of a number and then put it in the array at that index of number. For example, the number searched is 5 then it will put all the occurrence of the number at index five of the array.

**Pseudocode:**

**Counting-Sort(array, col-index)**

Let Col-Price = [val[col-index] for val in array]

Let Max-Element = Get maximum of (Col-Price)

Let Min-Element = Get minimum of (Col-Price)

Let range = Max-Element – Min-Element + 1

Let Count = array of zero of size of (range)

Let Output = array of zero of size of (array)

for i = 0 to array.length

Count[array[i][col-index]-Min-Element] +=1

for i= 1 to Count.length

Count[i] += Count[i-1]

for i= array.length down to 0

j = Count[array[i][col-index]-Min-Element] - 1

Output[j] = array[i]

Count[array[i][col-index]-Min-Element] - =1

return output

**Python Code:**

**def counting(self,Arry, col\_index) :**

col\_Price = [val[self.col\_index] for val in Arry]

Max\_Element = max(col\_Price)

Min\_Elemenet = min(col\_Price)

rang = Max\_Element - Min\_Elemenet + 1

Count = [0]\*(rang)

Output = [0]\*len(Arry)

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

Count[Arry[i][col\_index]-Min\_Elemenet] +=1

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

Count[i] += Count[i-1]

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

j = Count[Arry[i][col\_index]-Min\_Elemenet] - 1

Output[j] = Arry[i]

Count[Arry[i][col\_index]-Min\_Elemenet] -=1

return (Output)

**Time Complexity:**

Worst Complexity: O(n)

Average Complexity: O(n+k)

Best Complexity: O(n+k)

**Strength:**

Counting sort is the faster then all the comparison bases sorting algorithms.

**Weakness:**

Counting sort is only works for the integers. It sorts the string because it cannot make buckets of characters to strings there. It need integers to keep the count of the elements.

**Dry Run:**

**Searching Algorithms:**

**Linear Search:**

In this searching algorithm. The element is searching is started from the first index of the array. If the element is found before the last elements it returns the elements. If not, it will run until the last index of the array.

**Pseudocode:**

Linear-Search(array, element)

Let array1=[]

for i= 0 to array.length

if element==array[i]

append the element to array1

def linearSearch(array, element)

array = [ ]

for i in range (len(array)):

if element in array[i]:

print(array[i])

array.append(array[i])

**Time Complexity:**

**Worst Complexity:** O(n)

**Average Complexity:** O(n)

**Best Complexity**: O(1)

**Strength:**

The biggest advantage of the linear sort is that it can find the element even from the unsorted array. It works very good for small amount of data. It can find strings from the data also.

**Weakness:**

For large amount of data linear sort may take a lot of time to search an element.