Fashion Clothes Info

**Midterm Project**

A picture containing text, clipart

Description automatically generated

**Project Supervisor**

Mr. Samyan Qayyum Wahla

**Project Members(G16)**

|  |  |
| --- | --- |
| Wasfa Nasir | 2020-CS-145 |
| Aiemen Altaf | 2020-CS-148 |

Department of Computer Science

University of Engineering and Technology, Lahore Pakistan

Table of Contents

[Project Name: 3](#_Toc86854187)

[Motivation: 3](#_Toc86854188)

[Description: 3](#_Toc86854189)

[Technology Stack: 4](#_Toc86854189)

[Project Features: 5](#_Toc86854189)

[Business Case: 6](#_Toc86854190)

[Analysis of Project: 7](#_Toc86854195)

[Algorithms: 45](#_Toc86854196)

[Design 46](#_Toc86854197)

[Technical Details: 46](#_Toc86854198)

[Interfaces: 47](#_Toc86854199)

[Final GUI: 48](#_Toc86854200)

Testing/Issues[: 51](#_Toc86854201)

[Collaboration: 51](#_Toc86854202)

[Task Division: 52](#_Toc86854203)

[Issues: 54](#_Toc86854204)

# Project Name:

Fashion Clothes Info

# Motivation:

We can collect data at greater volume than a single human could ever hope to achieve.

**Description:**

In search of good quality and affordable clothes my time was wasted in exploring so many websites of different brands. I tried to find out when the best time to buy clothes, but I did not find any efficient way. So, I decided to build a small program to automatically collect the data from the website of my choice. It extracts information for my specific interest and displays the category, price, colours, and many more features of clothes I want to buy that I was searching. So, basically, Web Scraping is a technique used to extract data from websites through an automated process. Our project is about scrapping data of clothes with their attributes from websites and then putting them in a way that the data is sorted in well formatted form using different technique of Data Structures. In this project we use algorithms for data management. We are formulating the ideas of different algorithms and sorting techniques. All these techniques of algorithms are implemented on our data. For example: Types of sorting like merge sort, Insertion sort, Bubble sort, selection Sort and many more concepts we studied in DSA. With the help of all these techniques, we sort the attributes (data) in ascending and descending order up to user’s choice. We try to illustrate a progress bar that will display clearly that how much data has been scraped and if we want to pause or resume the scraping. Basically we are using different algorithms techniques and at the end we will find the time complexity taken by the whole process. To make UI interactive it is necessary to display data in such a format that is user friendly. Here, a user can see/buy any type of clothing product rather it’s for men, women, kids, bridal, winter summer, new in or all, etc.

# 

# Technology Stack:

|  |  |
| --- | --- |
| Language | Python |
| Platform | Web |
| Frontend Technology | pyQt designer |
| IDEs | Anaconda(6 Notebook) |

# 

# Project Features:

**List of Features:**

Implementing Insertion Sort on data

Implementing Selection Sort on data

Implementing Bubble Sort on data

Implementing Merge Sort on data

Implementing Quick Sort on data

Implementing Counting Sort on data

Implementing Bucket Sort on data

Implementing Radix Sort on data

Implementing Heap Sort on data

Implementing Slow Sort on data

Implementing Cycle Sort on data

Implementing Shell Sort on data

Implementing Comb Sort on data

Implementing Sleep Sort on data

# Business Case:

|  |  |
| --- | --- |
| Outline the business need for the project | Many of the clothing brands and websites have an easy access in searching the clothes with their brands and related prices according to latest fashion. This provides a benefit to clothing brands to save their time and increase their rating. |
| End user of the product | Our target is those Customers that want to buy products using online resources. |
| State the level of impact expected should the project proceed and implications of not proceeding | We can access and collect any data that is available on their website that we scrapped. To store data in efficient way and to save people’s time, this technique is fruitful in many ways. |

# Analysis of Project:

**Algorithm Name:**

1. Merge Sort

**Description:**

The merge filtering algorithm closely follows the divide-and-conquer paradigm. Divide and conquer is a way to divide a problem into a few small problems that are minor cases of the same problem. After that, we overcome the problems below by solving them repeatedly. However, if the size of the minor problems is small enough, just solve the problem in a simple, straightforward way. Combine small problem solutions into the solution of the first problem. This algorithm divides two identical groups into two halves and then combines two identical members. The main function of the composite type algorithm is to combine two sequential sequences in the "merge" step. We use the Merge Function MERGE (A p, q, r) when we assume that subarrays are arranged in order. It combines them to form a single filtered frame that replaces the current subarray.

**Pseudo Code:**

MERGE(A, p ,q, r):

n1 = q – p + 1

n2 = r – q

let L[1 . . n1 + 1]and R[1. . n2 + 1] be new arrays

for I = 1 to n2:

L[I] = A[p + I – 1]

for j = 1 to n2:

R[j] = A[q + j]

L[n1 + 1] = (infinity)

R[n2 + 1] = (infinity)

I = 1

j = 1

for k = p to r:

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

A[k] = L[i]

I = I +1

Else A[k] = R[j]

j = j+1

MERGE-SORT(A, p, r)

if p<r

q= [(p+r)/2]

MERGE-SORT(A, p, q)

MERGE-SORT(A, p+1, r)

MERGE(A, p ,q, r)

**Python Code**:

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

n1 = q – p + 1

n2 = r – q

#let L[1 . . n1 + 1]and R[1. . n2 + 1] be new arrays

L = arr[ :mid]

R = arr[mid:]

for I = 1 to n2:

L[I] = A[p + I – 1]

for j = 1 to n2:

R[j] = A[q + j]

L[n1 + 1] = (infinity)

R[n2 + 1] = (infinity)

I = 1

j = 1

for k = p to r:

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

A[k] = L[i]

I = I +1

Else A[k] = R[j]

j = j+1

def MERGE\_SORT(A, p, r):

if p < r:

q= [(p + r)/2]

MERGE\_SORT(A, p, q)

MERGE\_SORT(A, p+1, r)

MERGE(A, p ,q, r)

**Time Complexity Analysis:**

Best Case: O (nlogn)

Worst Case: O (nlogn)

Average Case: O (nlogn)

**Proof of Correctness:**

**Initialization**: As the algorithm starts by using divide and conquer and dividing the array into two halves. L[i] and R[j] are the smallest element of the left and right sub arrays.

**Maintenance:** Each iteration of the loop divides the left and right arrays and sort them accordingly by applying the Merge() function of each sub array.

**Termination:** The program will terminate when both left and right arrays are sorted in ascending or descending order and then we combine both arrays by using merge sort().

**Three Strengths:**

1. This algorithm access data with a slow rate as compared to others.
2. It is a stable Algorithm.
3. Those data which is in sequence, Merge Sort is best algorithm to apply for them.

**Three Weaknesses:**

1. This Algorithm is slower than other algorithms.
2. It requires an extra memory space.
3. This algorithm completes the whole process even if the array is sorted already.

**Dry Run:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 21 | 20 | 23 | 24 | 26 | 25 |

|  |  |  |
| --- | --- | --- |
| 21 | 20 | 23 |

|  |  |  |
| --- | --- | --- |
| 24 | 26 | 25 |

|  |
| --- |
| 23 |

|  |  |
| --- | --- |
| 21 | 20 |

|  |
| --- |
| 25 |

|  |  |
| --- | --- |
| 24 | 26 |

|  |
| --- |
| 20 |

|  |
| --- |
| 21 |

|  |
| --- |
| 24 |

|  |
| --- |
| 26 |

**Algorithm Name:**

1. Selection Sort

**Description:**

The selection algorithm includes a simple filtering algorithm around the comparison. It selects an object from every repetition and puts it in its proper place. This algorithm selects the smallest item from the list and places it at the beginning of the same members alternately. This can also be done in contrast to selecting the top feature.

**Pseudo Code:**

n is length and A is for array.

for i = 1 to n – 1

now select the current element as i

min = i

now check if the element is minimum

for j = i+1 to n

if A[j] < A[min]

min = j

Now swap the minimum element with the current element

swap A[min] with A[i]

**Python Code**:

def Selection\_Sort(A, n):

for i in range(n):

min=i

for j in range(i + 1, n):

if A[j] < A[min]:

min = j

(arr[i], arr[min]) = (arr[min], arr[i])

Print(“Sorted Array is:”)

A =[1,3,4,5,2,6]

n= len(A)

Selection\_Sort(A,n)

**Time Complexity Analysis:**

Best Case: O (n)

Worst Case: O (n^2)

Average Case: O (n^2)

**Proof of Correctness:**

**Loop Invariant:** A[min] is the smallest element

**Initialization:** Before the primary repetition of the loop, j = i + 1. So fifty per cent of the identical individual's A [i..j-1] are surely A [i]. Since line 2 of the code units min = I, we've got that min pointing to the smallest object (most effective one object) in subarray A [i..j-1] and ¬so loop compatibility is true.

**Maintenance:** Before passing j, we assume that the min points to a small object in the subarray A [i..j-1]. During repetition j we have two conditions: either A [j] <A [min] or A [j] ≥ A [min]. In the second case, the statement in line 4 is not true, so nothing is done. But now min points to the small object of A [i..j]. In the first case, line 5 changes the minute to point j as it is too small. If the minus indicates something less than or equal to subarray A [i..j-1] and still A [j] <A [min], then it should be the case that A [j] is less than or equal to elements in subarray A [i..j-1]. Line 5 changes minutes to point to this new location and so after completing the loop repetition, the min points to something very small in it subarray A [i..j].

**Termination:** When the inner loop is disconnected, the min identifies an object that is less or equal to all the elements in the lower column. From j = n + 1 when finished. This finds a very small object in this sub-program and is useful for us on the outer loop because we can move that very small object to the right place.

**Three Strengths:**

1. No more space is required that is why we don’t use linked list, array list etc.
2. It is a stable Algorithm.
3. Arrangement of elements do not affect its performance.

**Three Weaknesses:**

1. It is ideal for O(n^2) when n is small.
2. Its Time complexity is very poor.
3. Selection sort works less efficiently than Insertion Sort.

**Dry Run:**

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| 35 | 30 | 26 | 11 |

I=1

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| 11 | 30 | 26 | 35 |

I=2

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| 11 | 26 | 30 | 35 |

I=3

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| 11 | 26 | 30 | 35 |

**Algorithm Name:**

3.Counting Sort

**Description:**

Counting Sort is a sorting approach wherein an array is looked after primarily based totally on counting numbers. Counting type determines, for every enter detail x, the range of factors much less than x. It makes use of these records to area detail x without delay into its function within side the output array. We make a brand-new array and rely on numbers of a real array on this 2d array. Then we upload the indexes like in cumulative frequency. This array is once more thein examined with the real array after which the very last array carries the looked-after numbers. A vital asset of the counting type is that it's far stable: numbers with the identical cost seem withinside the output array withinside the identical order as they do withinside the enter array. That is, it breaks ties among numbers with the aid of using the guideline of thumb that whichever range seems first withinside the enter array seems first withinside the output array. Counting type’s balance is vital for some other reason: counting type is regularly used as a subroutine in radix type.

**Pseudo Code:**

function Counting-Sort(input)

k = range of elements of array

count ← array of k + 1 zeros

output ← array of same length as input

for i = 0 to length(input) - 1 do

j = key(input[i])

count[j] += 1

for i = 1 to k do

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

for i = length(input) - 1 down to 0 do

j = key(input[i])

count[j] -= 1

output[count[j]] = input[i] return output

**Python Code**:

def CountingSort(Num):

n=len(Num)

Res=[0]\*n

C=[0]\*10

for I in range(0,n):

C[Num[I]] =C[Num[I]]+1

for I in range(1,10):

C[I]=C[I]+C[I-1]

i=n=1

while i>=0:

Res[C[Num[i]]-1]=Num[i]

C[Num[i]] -= 1

i=i-1

return Res

Num = list(map(int,input().split()))

CountingSort(Num)

print(Num)

**Time Complexity Analysis:**

Best Case: O (n ) n is no. of elements in array

Worst Case: O (n )

Average Case: O (n )

**Proof of Correctness:**

**Initialization:** After the for loop of strains 2–three initializes the array C to all zeros, the for loop of strains 4–five inspects every entry detail. If the cost of an enter detail is I, we increment C[i]. Thus, after line five, C[i] holds the wide variety of entering factors same to I for every integer i – 0,1,…..,k.

**Maintenance**: Lines or Rows 7–8 determine how many 0,1,…., K are input elements less than or equal to i by keeping the active amount of the same members C.

**Termination: the for loop of lines 10–12 places each element AŒj into its correct sorted position in the output array B.**

**Three Strengths:**

1. This algorithm is stable.
2. It can also be used for negative inputs also.
3. Counting sort is often used as a subroutine in radix sort.

**Three Weaknesses:**

1. Counting sort only works if the no. of inputs is known.
2. If range of an input array is so large, then it requires a lot of space.
3. It only works by iterating through the input array.

**Dry Run:**

A: ( i )

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2 | 1 | 2 | 6 | 0 | 4 | 3 | 3 |

C:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 2 | 2 | 1 | 0 | 1 |

C:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 2 | 4 | 6 | 7 | 7 | 8 |

Result:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 2 | 3 | 3 | 4 | 6 |

Result[C[A[i]] – 1] = A[i]

C[A[i]] = C[A[i]] – 1

**Algorithm Name:**

1. Radix Sort

**Description:**

Radix Sort is a set of rules that divides digitally from the least crucial to the maximum crucial. Radix kind solves the cardboard configuration problem. For decimal digits, every column makes use of the simplest 10 places. (Two different places have been reserved for non-numeric characters.) The d-digit quantity will then update the d-column field. Radix filtering solves the cardboard-sorting problem — in contrast — through filtering the maximum crucial digit first. The set of rules then combines playing cards right into an unmarried deck, playing cards in zero bin precede playing cards in 1 bin precedent playing cards in 2-bin, and so on. It makes use of the calculation kind as a subset. For the radix to clear out to paintings properly, the digit kinds should be stable. The form of card clear out is stable, however, the operator should be cautious now no longer to extrude the order of the playing cards as they arrive out of the bin, even though all the playing cards are withinside the identical digit withinside the decided on the column.

**Pseudo Code:**

RADIX-SORT(A,d)

1. for i = 1 to d
2. Use a stable sort to sort array A on digit i usually counting sort.

**Python Code**:

def CountingSort(Num, P):

n = len(Num)

Res = [0]\*n

C = [0]\*10

for i in range(0, n):

temp = Num[i] // P

C[temp%10]+= 1

for i in range(1,10):

C[i] = C[i] + C[i-1]

i = n-1

while i >= 0:

temp = Num[i] // P

Res[C[temp%10]-1] = Num[i]

C[temp%10] -=1

i= i-1

for i in range(0,n):

Num[i] = Res[i]

def RadixSort(Num):

maximum= max(Num)

N = 1

while maximum // N > 0:

CountingSort(Num, N) #Counting sort algorithm

N= N\*10

Num = list(map(int,input().split()))

RadixSort(Num)

print(Num)

**Time Complexity Analysis:**

Best Case: O (d(n+k)

Worst Case: O ( d(n+k)

Average Case: O (d(n+k )

Each pass over n d-digit numbers then takes time O (n+k). There are d passes, and so the total time for radix sort is O (d(n+k )

**Proof of Correctness:**

**Loop invariant:** At the beginning of the loop, the same members are cared for with the remaining i-1 digits.

**Initialization:** Similar members are given less attention to the remaining last 0 digits.

**Maintenance:** Let's expect the system to take care of the remaining i-1 digits. After typing the appropriate digit, the same members may be cared for in the remaining ith digits. It is clear that the digital features of the same type within the space in which they are placed are properly arranged; within the same number of digits, however, we get the correct order, because it is the use of a solid type, and the features are already taken care of in the remaining i-1 digits.

**Termination:** The loop breaks when i = d + 1. Since consistency is gripping, we have numbers sorted by d digits.

**Three Strengths:**

1. It is used where there are long numbers like mobile numbers.
2. This algorithm is well known for its fastness for digits and even strings.
3. It is efficient algorithm for numbers in descending order.

**Three Weaknesses:**

1. It takes more space than Quick sort.
2. It can only be implemented on digits or the numbers we want digitally.
3. It is a slow process when apply on small set of data

**Dry Run:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 423 | 324 | 123 | 5 | 725 |

Total passes: 3

1st pass:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 42**3** | 32**4** | 12**3** | 00**5** | 72**5** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 42**3** | 12**3** | 32**4** | 00**5** | 72**5** |

2nd pass:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0**0**5 | 4**2**3 | 1**2**3 | 3**2**4 | 7**2**5 |

3rd pass:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **0**05 | **1**23 | **3**24 | **4**23 | **7**25 |

**Algorithm Name:**

5. Heap Sort

**Description:**

Heapsort combines the higher attributes of the 2 sorting algorithms Merge and insertion sort. We name it a “heap,” to control information. It additionally makes a priority precedence queue. It works as a entire binary tree search.

**Pseudo Code:**

PARENT(i)

return [i/2]

LEFT(i)

return 2i

RIGHT(i)

return 2i + 1

**Python Code:**

def Heap-Sort(A):

n= len(A)

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

heap(A, n, I)

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

A[I] , A[0] = A[0], A[I]

Heap( A, I ,0)

A= [1,12,4,6,3,7]

Heap-Sort(A)

**Time Complexity Analysis:**

Best Case: O (nlogn)

Worst Case: O (nlogn)

Average Case: O (nlogn)

**Proof of Correctness:**

**Three Strengths:**

1. The time required for this algorithm increases as the algorithm precedes.
2. No extra memory is used in this process.
3. It has basic computer science concepts, so it is simpler to use.

**Three Weaknesses:**

1. It does not has more applications like quick or merge sort.
2. It is not stable.
3. It’s time complexity is O(1)

**Dry Run:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

16

14

10

**Algorithm Name:**

6. Shell Sort

**Description:**

Shell Sort is a trendy model of Insertion sort. We first the elements that are far apart from each other in an array. By reducing the interval between them. It is based on a sequence of elements in an array.

**Pseudo Code:**

Shell-Sort(A, n)

For interval I < n/2 down to 1

For each interval “i” in array sort at interval “i”

End Shell-Sort

**Python Code**:

**def** Shell\_Sort(A):

     n **=** len(A)

**while** n> 0:

         i **=** 0

         j **=** n

**while** j < n:

**if** arr[i] >arr[j]:

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

             i = i +1

             j = j **+** 1

            k **=** i

**while** k **-** n > 1:

**if** arr[k **-** n] > arr[k]:

                    arr[k**-**n],arr[k] **=** arr[k],arr[k**-**n]

                k **=** k **–** 1

n **//=** 2

 A **=** [1,5,7,34,6,2]

Shell\_Sort(A)

**Time Complexity Analysis:**

Best Case: O (nlogn)

Worst Case: O (n^2)

Average Case: O (nlogn)

**Three Strengths:**

1. When insertion sort does not work due to far away spaces, shell sort helps to determive far apart elements.
2. Less number of swapping to elements is performed.
3. It is efficient for finite number of elements.

**Three Weaknesses:**

1. It is difficult to understand.
2. It is not stable because it do not determine the element in between the interval.
3. It’s slower than quick, merge or heap sort.

**Dry Run:**

**Algorithm Name:**

1. Insertion Sort

**Algorithm Description:**

Insertion sort is a simple sorting algorithm. This sorting is based on the idea of a sorted area in the list or array. Initially, the first key is considered to be the sorted part. Then the next key is compared with the first and is exchanged. Then the third key is considered relative to the first two keys and inserted; now the fourth key is considered and this process continues.

**Pseudo Code:**

INSERTION-SORT(A)

for i = 1 to n

key ← A [i]

j ← i – 1

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

A[j+1] ← A[j]

j ← j – 1

End while

A[j+1] ← key

End for

**Python Code:**

def insertionSort(A):

for step in range(1, len(A)):

key = A[step]

j = step - 1

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

A[j + 1] = A[j]

j = j - 1

A[j + 1] = key

A = [9, 5, 1, 4]

insertionSort(A)

print('Sorted Array:')

print(A)

**Time Complexity Analysis:**

Worst Case: O(n^2)

Base Case: O(n)

Average Case: O(n^2)

**Proof of Correctness:**

**Initialization:** The sub array starts with the first element of the array, and then starts to sort.

**Maintenance** – Each iteration of the loop expands the sub array, but keeps the sorted array properly. An element gets inserted into the array only when the element is greater than the element to its left. Since the elements to its left have already been sorted, it means it is greater than all the elements to its left, so the array remains sorted.

**Termination** - The program will terminate after reaching the last element in the array, which means the sorted sub array has expanded to encompass the entire array. Now the array is fully sorted.

**Three Strengths:**

1. Simplicity
2. Efficient
3. Stable

**Three Weaknesses:**

1. Its time complexity
2. Does not deal with huge list
3. Least advanced to other algorithms

**Dry Run:**

Initial Array is:

A = [9,5,1,4]

Key = 5

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 5 | 1 | 4 |
| 9 | 9 | 1 | 4 |
| 5 | 9 | 1 | 4 |

Key = 1

|  |  |  |  |
| --- | --- | --- | --- |
| 5 | 9 | 1 | 4 |
| 5 | 9 | 9 | 4 |
| 5 | 5 | 9 | 4 |
| 1 | 5 | 9 | 4 |

Key = 4

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 5 | 9 | 4 |
| 1 | 5 | 9 | 9 |
| 1 | 5 | 5 | 9 |
| 1 | 4 | 5 | 9 |

A is sorted!

**Algorithm Name:**

1. Bubble Sort

**Algorithm Description:**

Bubble filtering is a simple filtering algorithm that repeatedly moves through a list, comparing interdependent elements and when they are in the wrong sequence or in an unfiltered sequence they rotate. Passing through the list is repeated until the list is completely filtered.

**Pseudo Code:**

Bubble Sort(A)

for i=0 to n-i-1

for j=0 to n-i-2

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

Swap A[j] and A[j+1]

**Python Code:**

def bubbleSort(A):

for i in range(len(A)):

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

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

temp = A[j]

A[j] = A[j+1]

A[j+1] = temp

A = [23,16,10,11,20]

bubbleSort(A)

print('Sorted Array :')

print(A)

**Time Complexity Analysis:**

Worst Case: O(n^2)

Best Case: O(n)

Average Case: O(n^2)

**Proof of Correctness:**

**Loop invariant:** At the beginning of each loop of 2-4 strips, the sub-array A [j..n] contains elements that are actually present in A [j..n] before loading but differently and the first element A [j] is very small between them.

**Initialization**: Initially the sub-list contains only the last part A [n] ,which is a very small part of the subarray.

**Maintenance:** In each step we compare A [j] A [j] and A [j - 1] A [j − 1] and make A [j - 1] very small between them. After replication, the length of the subarray increases by one and the first thing is very small in the subarray.

**Termination**: The loop breaks when j = i. According to the loop invariant statement, the A [i] is the smallest between A [i..n] and A [i..n] consists of elements at the beginning of A [i..n] before loading.

**Three Strengths:**

1. Easy to understand
2. Easy to implement
3. Lesser memory is required

**Three Weaknesses:**

1. Code inefficient
2. Slower than many other sorts
3. Inefficient for large lists

**Dry Run:**

A = [23,16,10,11,20]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 23 | 16 | 10 | 11 | 20 |

i=0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 10 | 16 | 11 | 20 | 23 |

i=1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 10 | 16 | 11 | 20 | 23 |

i=2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 10 | 11 | 16 | 20 | 23 |

i=3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 10 | 11 | 16 | 20 | 23 |

i=4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 10 | 11 | 16 | 20 | 23 |

A is sorted!

**Algorithm Name:**

1. Quick Sort

**Algorithm Description**:

Quick Sort is one of the most efficient sorting algorithms. It is primarily based on the splitting of an array into smaller ones and swapping (exchange) based on the comparison with 'pivot' element selected. Because of this reason, quick sort is also known as "Partition Exchange" sort. Like Merge sort, Quick sort also falls into the category of divide and conquer approach of problem-solving method.

**Pseudo Code:**

QUICKSORT(A, low, high)

{

if (low < high)

{

pivot\_index = partition(A, low, high);

quickSort(A, low, pivot\_index - 1); // Before pivot\_index

quickSort(A, pivot\_index + 1, high); // After pivot\_index

}

}

**Python Code:**

def partition(A,low,high):

i = ( low-1 )

pivot = A[high]

for j in range(low , high):

if A[j] <= pivot:

i = i+1

A[i],A[j] = A[j],A[i]

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

return ( i+1 )

# Function for Quick sort

def quickSort(A,low,high):

if low < high:

pi = partition(A,low,high)

quickSort(A, low, pi-1)

quickSort(A, pi+1, high)

A= [10, 80, 30, 95, 40, 50, 70]

n = len(A)

quickSort(A,0,n-1)

print ("Sorted array is:")

print(A)

**Time Complexity Analysis:**

Worst Case: O(n^2)

Best Case: O(n log(n))

Average Case: O(n log(n))

**Proof of Correctness:**

**Loop invariant:**

* All entries in p<=k <=I are ≤ pivot.
* All entries in A[i + 1 . . j − 1] are > pivot.
* If k = r then, A[r] = pivot.

**Initialization:** Before starting the loop, all the entries in the loops invariant or arrays are already satisfied as the two subarrays are empty A[i] , A[i+1 , j-1] and r is pointing the pivot.

**Maintenance:** While the loop is running, when A [j] is less than or equal to pivot, A [j] and A [i + 1] are swapped and i and j are enlarged. If A [j]> is a pivot, then increase only j.

**Termination:** After the loop ends, if j=r then all the elements in A are sorted.

**Three Strengths:**

1. Fast
2. Efficient
3. Deal with large list

**Three Weaknesses:**

1. Not stable
2. Slower than radix sort
3. When list is sorted, it shows horrible results

**Dry Run:**

A= [10, 80, 30, 95, 40, 50, 70] pivot

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 10 | 80 | 30 | 95 | 40 | 50 | 70 |

|  |  |
| --- | --- |
| 90 | 80 |

|  |  |  |  |
| --- | --- | --- | --- |
| 10 | 30 | 40 | 50 |

|  |
| --- |
| 90 |

|  |  |  |
| --- | --- | --- |
| 10 | 30 | 40 |

|  |  |
| --- | --- |
| 10 | 30 |

|  |
| --- |
| 10 |

**Algorithm Name:**

1. Cycle Sort

**Algorithm Description:**

Cycle kind is a risky set of rules. It is an assessment sorting set of rules that forces arrays to be factored into the wide variety of cycles in which every one of them may be circled to supply a looked after array. It is primarily based totally on the concept that the array to be looked after may be divided into cycles.

**Pseudo Code:**

CycleSort(A)

for a= 0 to n – 2 do

key = A[a]

location = a

for i = a + 1 to n-1

if A[i] < key

location=location +1

if location = a

while key = A[location] do

location = location +1

if location != a

swap A[location] and key

while location != a

location = a

for i = a + 1 to n-1 do

if A[i] < key then

location=location +1

while key = A[location]

location = location +1

if key != A[location]

swap A[location] and key

**Python Code:**

def cycleSort(A):

writes = 0

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

key= A[a]

location = a

for i in range(a + 1, len(A)):

if A[i] < key:

location += 1

if location == a:

continue

while key == A[location]:

location += 1

A[location], key = key, A[location]

writes += 1

while loaction != a:

location = a

for i in range(a + 1, len(A)):

if A[i] < key:

location += 1

while key == A[location]:

location += 1

A[loaction], key = item, A[location]

writes += 1

return writes

A = [3, 2, 1, 4, 6, 5]

n = len(A)

cycleSort(A)

print("After sorting : ")

for i in range(0, n) :

print(A[i], end = \' \')

Time Complexity Analysis:

Worst Case: O(n^2)

Base Case: O(n^2)

Average Case: O(n^2)

Proof of Correctness:

Three Strengths:

1. Not additional storage

Three Weaknesses:

1. Unstable

**Dry Run:**

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

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

1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 2 | 3 | 4 | 6 | 5 |

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

6

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

5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 6 | 6 |

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

A is sorted!

**Algorithm Name:**

1. Comb Sort

**Algorithm Description:**

Comb Sort is specifically a development over Bubble Sort. Bubble kind constantly compares adjoining values. So all inversions are eliminated via way of means of one. Comb Sort improves on Bubble Sort via way of means of the use of an opening of length greater than 1. The hole begins off evolved with a big price and shrinks via way of means of an element of 1. three in each generation till it reaches the price 1. Thus Comb Sort gets rid of multiple inversion depending on one switch and plays higher than Bubble Sort.

**Pseudo Code:**

combsort(A)

gap = input.size

shrink = 1.3

sorted := false

loop while sorted = false

gap = floor(gap / shrink)

if gap ≤ 1 then

gap = 1

sorted = true

end if

i = 0

loop while i + gap < input.size

if input[i] > input[i+gap] then

swap(input[i], input[i+gap])

sorted = false

end if

i = i + 1

**Python Code:**

def updateGap(gap):

gap = int((gap \* 10)/13)

if gap < 1:

return 1

else:

return gap

def combSort(A, n):

gap = n

swapped = True

while gap !=1 or swapped == 1:

gap = updateGap(gap)

swapped = False

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

if A[i] > A[i + gap]:

A[i], A[i + gap]=A[i + gap], A[i]

swapped = True

A= [ 7, 5, 1, 50, 3, -20, 25, -4, 30, 0]

n= len(A)

combSort(A, n)

print ("Sorted array")

for i in range(n):

print (A[i],end=' '

Time Complexity Analysis:

Worst Case: O(n^2)

Base Case: O(n^2/2^p),where p is no. of increments

Average Case: O(n logn)

**Proof of Correctness:**

The main loop when set = is false, and there are no break statements that stop the loop early, which is very helpful in proof: if that loop ever breaks, it should be the default method = true. So we have to show only two things: first that the loop is terminated, and second that if it is filtered = true at the end of the loop and then the list is actually sorted.

The loop breaks because each time the filter is still false at the end of the multiplication, the gap becomes smaller, or the gap remains the same and the amount of flexibility in the system becomes smaller. Both the gap and the conversion value are whole numbers tied below 1 and 0 respectively, so they cannot be infinitely smaller.

**Three Strengths:**

1. Simple
2. Efficient
3. Better than bubble sort

**Three Weaknesses:**

1. It takes time

**Dry Run:**

Initial array (7, 5, 1, 50, 3, -20, 25, -4, 30, 0)

Initial Gap value = 10

Constant k = 1.3

1st pass:

After update, gap value = 10/1.3 = 7

7 5 1 50 3 -20 25 -4 30 0

-4 5 1 50 3 -20 25 7 30 0

-4 5 0 50 3 -20 25 7 30 1

2nd pass:

After update, gap value = 7/1.3 = 5

-4 5 0 50 3 -20 25 7 30 1

-20 5 0 50 3 -4 25 7 30 1

-20 5 0 30 3 -4 25 7 50 1

-20 5 0 30 1 -4 35 7 50 3

3rd pass:

After update, gap value = 5/1.3 = 3

-20 5 0 50 3 -4 25 7 30 1

-20 3 0 50 5 -4 25 7 30 1

-20 3 -4 30 5 0 25 7 50 1

-20 3 -4 25 5 0 30 7 50 1

-20 3 -4 25 5 0 1 7 50 30

4th pass:

After update, gap value = 3/1.3 = 2

-20 3 -4 25 5 0 30 7 50 1

-20 3 -4 0 5 25 1 7 50 30

-20 3 -4 0 1 25 5 7 50 30

-20 3 -4 0 1 7 5 25 50 30

5th pass:

After update, gap value = 2/1.3 = 1

-20 3 -4 0 1 7 5 25 50 30

-20 -4 3 0 1 7 5 25 50 30

-20 -4 0 3 1 7 5 25 50 30

-20 -4 0 1 3 7 5 25 50 30

-20 -4 0 1 3 5 7 25 50 30

-20 -4 0 1 3 5 7 25 30 50 Array Sorted!

**Algorithm Name:**

1. Slow Sort

**Algorithm Description:**

Slowsort is a hard and fast of clear out filters. It is hard and fast of hesitant recommendations primarily based totally on the precept of repetition and dedication. Sort a couple of numbers from low to unequal (or unbalanced to low). We are given numerous numbers and we want to place them inside the ideal order. The enter approach set of the recommendations works as follows: We can restoration the hassle of sorting n numbers in series up to:

1. Find the most variety of numbers
2. Find the most variety of factors of the primary variety n / 2
3. locate the amount of the very last n / 2 factors
4. locate the super cost of those figures 5) very last editing

**Pseudo Code:**

slowsort(A, i, j)

if i ≥ j then

return

m = floor( (i+j)/2 )

slowsort(A, i, m)

slowsort(A, m+1, j)

if A[j] < A[m] then

swap A[j] , A[m]

slowsort(A, i, j-1)

**Python Code:**

def slowsort(A, i, j):

if( i ≥ j):

return m = floor( (i+j)/2 )

slowsort(A, i, m)

slowsort(A, m+1, j)

if (A[j] < A[m]):

A[m],A[j] =A[j] , A[m]

slowsort(A, i, j-1)

**Time Complexity Analysis:**

Worst Case: O(n^2)

Base Case: O(n)

Average Case: O(n^2)

**Algorithm Name:**

1. Bucket Sort

**Algorithm Description:**

Bucket Sort is a sorting algorithm .It is a distribution sort that works by distributing the array's element into several groups called buckets. Each bucket is then sorted individually by using any of the suitable sorting algorithms or recursively applying the same bucket algorithm.

**Pseudo Code:**

BucketSort(A)

n = A.length

Let B[0, . . . , n − 1] be a new array

for i = 0 to n - 1

B[i] ← 0

for i = 1 to n

B[bnA[i]c] ← A[i]

for i = 0 to n-1

sort list B[i] using insertion sort

concatenate the lists B[0], B[1], . . . , B[n − 1]

return B

**Python Code:**

def bucketSort(A):

bucket = []

for i in range(len(A)):

bucket.append([])

for j in A:

index\_b = int(10 \* j)

bucket[index\_b].append(j)

for i in range(len(A)):

bucket[i] = sorted(bucket[i])

k = 0

for i in range(len(A)):

for j in range(len(bucket[i])):

A[k] = bucket[i][j]

k += 1

return A

A = [0.40, 0.30, 0.20, 0.50, 0.21, 0.42, 0.53]

print("Sorted Array is :")

print(bucketSort(A))

**Time Complexity Analysis:**

Worst Case: O(n^2)

Best Case: O(n + k)

Average Case: O(n + k)

**Proof of Correctness:**

To see if this algorithm works, consider two aspects A [i] and A [j]. Think without the usual loss that A [i] <= A [j]. From nA [i] <= nA [j], either the element A [i] goes to the same bucket as A [j] or enters the bucket with a lower index. When A [i] and A [j] enter the same bucket, the 7–8 rows of rows place them in the correct order. If A [i] and A [j] go into different buckets, then line 9 puts them in the correct order. Therefore, the bucket arrangement works well.

**Three Strengths:**

1. Stable
2. Work in linear time
3. Fast

**Three Weaknesses:**

1. Cannot apply all data types
2. Its efficiency is sensitive
3. Its performance depends upon no. of bucket chosen

**Dry Run:**

A = [0.40, 0.30, 0.20, 0.50, 0.21, 0.42, 0.53]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0.40 | 0.30 | 0.20 | 0.50 | 0.21 | 0.42 | 0.53 |

This is array(empty) is used as a bucket

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Insert elements from array to bucket

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0.40 | 0.30 | 0.20 | 0.50 | 0.21 | 0.42 | 0.53 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0.20  0.21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Similarly for all

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0.20  0.21 | 0.30 | 0.40  0.42 | 0.50  0.53 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Sort the elements

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0.20  0.21 | 0.30 | 0.40  0.42 | 0.50  0.53 | 0 | 0 | 0 | 0 |

From each bucket elements are gathered

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0.20  0.21 | 0.30 | 0.40  0.42 | 0.50  0.53 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0.20 | 0.21 | 0.30 | 0.40 | 0.42 | 0.50 | 0.53 |

A is sorted!

**Algorithms:**

|  |  |
| --- | --- |
| Searching Algorithms | ***Linear search:***  *Linear search is a search that finds an element in the list by searching the element sequentially until the element is found in the list.*  ***Binary search****:*  *Binary search is a search that finds the middle element in the list recursively until the middle element is matched with a searched element.* |
| Searching Filters for each data type | *1 .GUI should have the option for searching based on each column.*  *2. User has choice to choose any algorithm for sorting for a particular column.*  *3. Advanced filters for string columns are used such as contains, end with, starts with etc.* |
| Multi-Level Sorting | *GUI should have the option for sorting of each column. So user have option to sort data using multiple columns.* |
| Any other features | *Scrapping task has the option to pause, start, resume and stop.* |

**Design:**

|  |  |
| --- | --- |
| ***Technical Details*** |  |
| Name of Entity | *Clothes* |
| Attributes of Entity  (Minimum seven attributes/rows can be increased) | |  |  |  | | --- | --- | --- | | *Name* | *Data Type* | *Description* | | *Clothes’ ID* | *Integer* | *Brand no. of clothes* | | *Category* | *String* | *Category of clothes like men, women, kids* | | *Collection* | *String* | *Collection of clothes i.e. summer, winter, new arrival.* | | *Brand* | *String* | *Name of brand* | | *Size* | *String* | *Size of clothes like Small, Medium, Large* | | *Combination* | *String* | *Combination of Clothes i.e.1pc, 2pc, 3pc* | | *Price* | *String* | *Different prices of clothes from low to high.* | |  |  |  | |
| Sample of Scrapping Source | Description: Graphical user interface, website  Description automatically generated  Description: Graphical user interface, text, email  Description automatically generated |
| Github Repository Link | *https://github.com/Aiemenaltaf/CS261F21PID56* |
| Sorting Algorithms | *Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Quick Sort, Counting Sort, Bucket Sort, Radix Sort, Heap Sort, Slow Sort, Cycle Sort, Shell Sort, Comb Sort, Sleep Sort* |

|  |  |
| --- | --- |
| ***Interfaces for your project*** |  |
| *[Draw layouts in the pencil tool. For each picture of the UI, provide the following table.]*  Description: Graphical user interface, text, application, email  Description automatically generated  Description: Table  Description automatically generated  Description: Graphical user interface, application, Word  Description automatically generated   |  |  |  | | --- | --- | --- | | UI Component Name | Type of UI component | Purpose of UI Component/Other details | | *Sorting Algorithms* | *Button* | *To select those algorithms which we want to implement* | | *Algorithms* | *Drop-down list* | *List of Algorithms to select one Algorithm.* | | *Search* | *Button* | *To search specific algorithm.* | | *Side bar* | *Scroll bar* | *Used to scroll the screen* | | *Data Scrapped* | *Button* | *To show how much data has been scrapped* | | *Bar* | *Progress bar* | *To show how much data has been scrapped* | | Clothes’ ID | *Button* | *To show list of Clothing Id’s* | | Category | *Button* | *To show list of Clothing Category* | | Collection | *Button* | *To show list of Clothing Collection* | | Brand | *Button* | *To show list of Clothing Brand* | | Size | *Button* | *To show list of Clothing Size* | | Combination | *Button* | *To show list of Clothing Combination* | | Price | *Button* | *To show list of Clothing Price* | | Time Complexity | *Button* | *To show how much time complexity is used by each algorithm.* | | Main picture | *Picture* | *Main picture on the screen.* | | |

**Final GUI:**

**Graphical user interface, website

Description automatically generated**

Final GUI using Pyqt5 designer app

**Testing/Issues:**

**Collaboration:**

Collaboration has always remained a major task in groups doing projects. Students find it difficult to collaborate with their companions. This may be due to some time management issues or members are not compatible with each other due to some reasons. In our situation, I and my group member are unanimous in every aspect of the project. We both understand the criteria of every deadline and tried our best to meet so. Time management was fully organized that if one member was working on the project the other member on the other hand was also putting the same efforts on it. We discuss every task before implementing it and during discussion, we respect each other opinions. It was always a healthy discussion between us. There is a myth that in a group of two, one is in leading position and the other works according to that person but we as I said before, respect each other point of views, so we both take the lead in our hands as compatibility is the major priority in a fruitful group work. We both worked hard on this project. We both put equal efforts on every task either they are physical or mental. We tried our best to give our 100%. The best part was that besides all the efforts, we both uplift each other in every problem we face while doing this project. To conclude, our collaboration was very good throughout the whole project.

**Task Division:**

Every task assigned in each deadline was equally distributed between 2020-CS-145 and 2020-Cs-148. We both worked accordingly on each task whether it’s project proposal or scrapping. Detail of the tasks and their division among the two members is given below:

|  |  |  |
| --- | --- | --- |
| Tasks | 2020-CS-145 | 2020-CS-148 |
| Project Proposal | 50% of the proposal is done by CS145. Algorithm’s detail, summary and some other fields are filled by 145. | Other 50% is done by CS148. This includes Interface, some technical details, and some other fields. |
| Graphic User Interface | GUI was made by both members and then we finalize the GUI in a single document. The software we used was pyqt5 designer tool. We first setup our python path and then install the designer. | GUI was made by both members and then we finalize the GUI in a single document. The software we used was pyqt5 designer tool. We first setup our python path and then install the designer. |
| Scrapping | Scrapping was done by both members. We both setup the scrapping setup in Anaconda jupyter notebook. Libraries we use were selenium, beautifulsoup4, pandas. Webdriver is also used to do a safe scrapping from any website. We both scrapped data so that good results could be drawn. Website we used if flipkart. | Scrapping was done by both members. We both setup the scrapping setup in Anaconda jupyter notebook. Libraries we use were selenium, beautifulsoup4, pandas. Webdriver is also used to do a safe scrapping from any website. We both scrapped data so that good results could be drawn. Website we used if flipkart. |
| Integration | Integration was a difficult task. We both did it accordingly. Firstly, we convert pyqt5 designer file to executable file (.py). After this we load data from csv file which was generated when we did scrap. Show this csv data into pyqt5 designer table. Table on GUI we made was showing data that was in our GUI. After this we apply algorithms to the code. Data in csv was sorted by using algorithms we used in the code. | Integration was a difficult task. We both did it accordingly. Firstly, we convert pyqt5 designer file to executable file (.py). After this we load data from csv file which was generated when we did scrap. Show this csv data into pyqt5 designer table. Table on GUI we made was showing data that was in our GUI. After this we apply algorithms to the code. Data in csv was sorted by using algorithms we used in the code. |
| Sorting Algorithms | This task was equally divided. CS145 did first 6 algorithms which include Merge sort, Selection sort, Counting sort, Radix Sort, Heap Sort, Shell Sort. | This task was equally divided. CS148 did other 7 algorithms which include Insertion sort, Quick sort, Bucket sort, Bubble Sort, Comb Sort, Cycle Sort, Slow Sort. |
| Final Report | Final Report is done by both group members. We completed our final report in university by doing work together in library. In final Report, there are proper contents of a report that include requirements, analysis of project, design, collaboration, task division, and issues. Other details are added in the document. | Final Report is done by both group members. We completed our final report in university by doing work together in library. In final Report, there are proper contents of a report that include requirements, analysis of project, design, collaboration, task division, and issues. Other details are added in the document. |

**Issues/Problems:**

* As said earlier that there is no such problem which do not have a definite solution. Yes, we face such problems during projects which at that time create such ambiguous situations. Scrapping seems an arduous task for us. We were working on a website named laam.official link is (<https://laam.pk/pages/brands>). This website has many sub classes and different brands were running at one site, so it was very difficult for us to find appropriate classes. So, we moved to daraz.pk link is (<https://www.daraz.pk/wow/gcp/daraz/channel/pk/dmart/dmart-choose-your-location/>). The problem here was total pages shown were around 125 but data was only present on 30 pages. Rest of the pages were blank. Also, some products were repeated in every page. So, we decided to move to flipkart link is (<https://www.flipkart.com/>) as data scrapped was not fulfilling our requirement. Now, we were happy as there were around 3257 pages and around 2 lacs plus products were available. But the same problem occur here as data was only on 25 pages, rest of the pages were encrypted. In this whole scenario the time wastage was so much because we know how to code in scrapping but could not find a good website! This was the major issue we faced in the whole project.
* We both never used late submission, so we have not much idea about late submission time. We decided to submit Algorithm’s report in late submission. Unfortunately, CS145 was attending a close family wedding in those days and CS148 was very ill. But we hardly managed to complete the report but faced submission problems first time.
* Integration was a difficult task for all of us. Time was very limited, and we have not how known that how to use scrapped data in algorithms or show on table. We tried a lot, but results were not so productive. Also, there was no one in university from whom we can take help as time was limited and all other people were also busy. We have no guidance for this task from anyone. So, this task was difficult for us.
* Loading data creates much ambiguity in the project like we wasted our whole 2 days in that. All the work has to be done by our own, so it was a like a tough mind game for us. But as we said before, that if you determine to do something at any cost, you will do that aside what will happen or how much short time you have.

This project seems an experience in which we learn a lot. Yes, we put a lot of effort in it. We face much problem, time was a little bit short, and we had to manage every task sincerely. We had to explore much and think more! But we believe that fruits of human labor do come. IN SHA ALLAH.

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