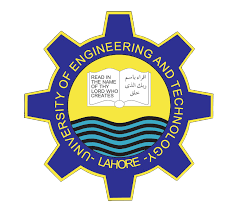
****University of Engineering and Technology

**WHAT MOBILE**

Desktop Application

**Group Members**

2020-CS-76

2020-CS-86

**Group ID**

CS261F21PID02

**Department of Computer Science**

**Instructor:** Samyan Qayyum Wahla

**Table of Contents**

* + - * + Objective
        + Description
        + Business Case
        + Learning outcome
        + Features
        + Details of features
        + Project requirements
        + Details of Algorithms
        + Attributes on Entity
        + Technical Details
        + Technology stack
        + Use cases
        + UI component detail
        + Project Planning

**Objective**

Mobile phone is an essential need of every person nowadays and everyone wants a efficient phone with low price and high specification so that one can enjoy the technology of modern times, so we have designed an application in which you can find different models of mobile phone and their specification along with their price mentioned, the user can be facilitated by aur application to buy his/her mobile phone as the data is scrapped from different websites and after sorting the data they can see which website is offering the best deal and the URL of the website is also given so that user is just one click away to buy his/her desired phone in efficient and less tiring manner.

**Description**

Our desktops application What Mobile will initially start with some data of phones within it which have been taken from a file. Different sorts of function can be applied to the given data. The user can sort the data in ascending order and descending order on every attribute individually. The user can also search for his desired phone from the given options that can also perform specific task on each column individually, the given options are such as is equal to, not equal to which can also be performed with another operator like OR, AND. The data in the table will be sorted with the help of the input given by the user.

As the sorting of the data can be done on each column individually but it can be done by nine different methods(algorithms) and each method will take different time for sorting, so an option (time taken) is given so that the user can see which method is better to sort the data.

There is an option given in the dataset window for scrapping a huge amount of data with a start/stop button, by clicking on it, the data will be fetched from a file containing a huge amount of data about phones and a progress bar to show how much data is fetched, once the progress bar has finished the dataset window will be opened to show the data in table but if the user presses the start/stop button before the completion of progress bar, the fetching of the data will be stopped and you can see the unfinished fetched data by clicking on the exit button.

**Business Case**

Need for Project

Statistics show that the majority of all online shoppers use reviews to determine what products to buy and which services to purchase.

Reviews not only have the power to influence consumer decisions but can strengthen a company's credibility. Reviews have the power to gain customer trust, and they encourage people to interact with the company. Customer interaction ultimately leads to improved profits for business.

End user of Product

Every Single Smart Phone user who wants to know about the latest mobile phones in the market can be facilitated by our app. As this application can be less tiring and time saving app for the users.

Motivation

The report presented here aims to identify strengths and weaknesses and expose lessons learned from what was a broadly successful experience and focusing on ways to organize a typically long-lived corpus of data so that queries can be answered really fast.

Level of impact

Our project will be helpful for general user to tell them which website is giving authentic result and offering a good package to buy phones. If our project is implications of not proceeding then we will update the project according to the requirements

**Learning outcomes:**

* The use of pyqt5
* How to scrap over million data from websites
* Save and extract a huge amount of data from csv file
* Perform multiple searching task using different conditions on data
* Perform sorting on a data using individual column

**Features**

Sorting Order:

* Ascending Order
* Descending Order

Search Algorithm:

* Binary Order
* Linear Order

Filters:

* Equal to
* Not Equal to

**Feature Details**

Sorting Order

**Ascending Order:** This filter will sort the data in lower to higher form.

**Descending Order:** This filter will sort the data in higher to lower form.

Searching Algorithms

**Linear Search:** It will start form the initial element of the data, after that each element is examined once its match is found and then the search ends.

**Binary Search:** This search algorithm will continue to divide the list into half until the desired element has been separated.

**Equal to:** This filter will allow us to find the desired value which is equal to the input using above search algorithms.

**Not Equal to:** This filter will allow us to find the desired value which is not equal to the input using above search algorithms.

**Multilevel Searching:** For multilevel searching user has been given the options AND,OR to search data.

**Requirements**

Sorting Algorithms

These are the names of the sorting algorithms which will be used in our application to sort data:

* Insertion Sort
* Selection Sort
* Bubble Sort
* Merge Sort
* Quick Sort
* Radix Sort
* Counting Sort
* Bucket Sort
* Tim Sort

User will have the choice to sort the data using any algorithms of a particular column.

**Details of Algorithm**

|  |  |
| --- | --- |
| **Name of Algorithm** | **Description** |
| Insertion Sort | Sorting by insertion is a relatively simple algorithm for building the final sorted array of items one by one. It also performs worse than other sorting algorithms on large lists. |
| Selection Sort | A selection sort is a method of sorting using in-place comparisons. Besides having a large O(n2) time complexity, it also performs worse than insertion sort on larger lists |
| Bubble Sort | Bubble sort consists of repeatedly stepping through a list, comparing adjacent elements, and swapping those that are out of order. |
| Merge Sort | In merge sort, the input array is split into two halves, each of which is sorted separately, then merged again. It is also sometimes referred as Stable sort algorithm. |
| Quick Sort | Quick Sort Algorithm works on divide and conquer technique. The main working step of this algorithm is to select pivot. Pivot selection are of the following types:   * Pick First or Last element as pivot. * Pick median as pivot. * Pick any arbitrary element as pivot.   Partition plays an important role in Quick Sort Algorithm. |
| Radix Sort | In Radix sorting, keys that share a similar significant position and value are grouped by individual digits, which then make up the keys. To sort an array of numbers, Radix sort uses counting sort as a subroutine. |
| Counting Sort | Counting sort uses keys between a specific range in order to sort. Objects are sorted by counting the number of key values that are different. |
| Bucket Sort | In Bucket sort an array of elements is sorted by bucketing them into a number of buckets, after that each bucket is sorted individually. |
| Tim Sort | Tim sort has been derived from insertion sort and merge sort which performs on data based on real world. The algorithm finds the data that is already in ordered form and then uses ordered ones to sort the remaining. |

**Insertion Sort**

**Description**

Insert sorting is a sorting mechanism in which the sorted array is created one by one. The elements of an array are sequentially compared and then arranged. Start with 1 unordered item on the left and N-1 on the right. Remove the first unordered array (item 2) and add the items to the sorted array by wrapping them as needed. We now have an ordered list of unordered items of dimensions 2 and N-2. Repeat for all items. It’s a bit like a bubble sort of moving objects, except it stops when you find an object smaller than you. If you sort the data backwards, each item should be at the top of the list, which will be a bubble sort. You can move an item to the left in several ways: you can replace each iteration, or you can copy all items to your neighbor.

|  |  |
| --- | --- |
| Pseudo code | int start  int insert  for doesn’t in range(1,len(ARR)):  insert = ARR[i]  start = i  while start > 0 and ARR[start-1] > insert:  ARR[start] = ARR[start -1]  start = start -1  ARR[start] = insert |
| Code in Python | def insertionSort(*arr*):  *for* I *in* range(1,len(*arr*)):          key = *arr*[i]          j = i-1  *while* j >= 0 and *arr*[j] > key:  *arr*[j+1] = *arr*[j]              j = j-1  *arr*[j+1] = key |
| Time complexity | **Best:** O(n)  **Worst:** O(n^2) |
| Dry Run | input:  3 1 0 15 7  i=0  3 1 0 15 7  i=1  1 3 0 15 7  i=2  0 1 3 15 7  i=3  0 1 3 15 7  i=4  0 1 3 7 15 |

**Proof of correctness**

A lot of proof techniques can be used, such as induction proof and contradiction proof can be used to prove the correctness but we will use "loop invariants". The loop invariant we will use is

we need to check three conditions:

* Initialization
* Maintenance
* Termination

**Initialization:** The first iteration, the sub array [1 to n - 1] is only the first element of the array, A [1]. This sub array is sorted and consists of the elements that were originally in A [1to 1].

**Maintenance:** Suppose that a [1 to n - 1] is sorted. The body of the for loop works until the correct position is found A [n], at which you insert the value of A [n]. The sub-array A [1 to n] is then made up of the elements at the origin of A [1 to n], but in an order. Increasing n for the next iteration of the for loop keeps the loop invariant.

**Termination:** The condition that causes the for loop to terminate is that n> total numbers. Since each iteration of the loop increases n by 1, we must have n = n + 1 at that time. We have shown that the sub-array A [1 to n] consists of the elements in sorted order.

**Strengths**

* It is easy to understand and is very clear.
* It works very fast for a small array.
* Insertion sort requires less space.

**Weaknesses**

* Downside to insertion does not perform well.
* Slows down when the amount of data is huge.
* Does not perform well on multiple items.

**Selection Sort**

**Description**

A selection sort is a method of sorting using in-place comparisons. Besides having a large O(n2) time complexity, it also performs worse than insertion sort on larger lists. In selection sort data is sorted by selection on conditions in start the data sorted array is empty and unsorted is in the right side hence divide so it sorts all data by selecting some entry.

|  |  |
| --- | --- |
| Pseudo Code | Def SelectionSort(arr):  arr = len(A)  n = len(arr)  for i = 1 to n – 1 do:  min = i  for j = i+1 to n  if arr[min] > arr[j]:  min = j;  swap arr[min] and arr[i] |
| Code in Python | def SelectionSort(*arr*):  *for* i *in* range(len(*arr*)):          min = i  *for* j *in* range(i+1, len(*arr*)):  *if* *arr*[min] > *arr*[j]:                  min = j  *arr*[i], *arr*[min] = *arr*[min], *arr*[i] |
| Time Complexity | **Best:** O(n^2)  **Worst:** O(n^2) |
| Dry Run | input:  3 20 0 13 7  i=0  3 20 0 13 7  i=1  0 20 3 13 7  i=2  0 3 20 13 7  i=3  0 3 7 13 20  i=4  0 3 7 13 20 |

**Proof of correctness**

A lot of proof techniques can be used, such as induction proof and contradiction proof can be used to prove the correctness but we will use "loop invariants". The loop invariant we will use is

we need to check three conditions:

* Initialization
* Maintenance
* Termination

**Strength**

* When the data is small it performs well.
* Due to in place algorithm there is no need of extra storage.

**Weakness**

* When there is huge amount of data that is to be sorted than the selection sort can take more time.
* The efficiency of selection sort is affected by the order of the initial list, if the data is randomly entered it will perform well.
* To sort n values, it will take n^2 time.

**Merge Sort**

**Description**

In merge sort, the input array is split into two halves, each of which is sorted separately, then merged again. It is also sometimes referred as Stable sort algorithm. Merge Sort uses divide and conquer rule as merge sort divides the given data into two portions and sorting these portions and recursively repeat this until the given data is sorted.

|  |  |
| --- | --- |
| Pseudo Code | Def mergesort( Array ):  if ( n == 1 )  return Array  left as array = a[0] ... a[n/2]  right as array = a[n/2+1] ... a[n]  left = mergesort( left )  right = mergesort( right )  return merge( left, right )  merge( a, b )  while ( a and b have elements )  if ( a[0] > b[0] )  add b[0] to the end of c  remove b[0] from b  else  add a[0] to the end of c  remove a[0] from a  end if  end while  while ( a has elements )  add a[0] to the end of c  remove a[0] from a  end while  while ( b has elements )  add b[0] to the end of c  remove b[0] from b  end while  return c |
| Code in Python | def mergeSort(*arr*):  *if* len(*arr*) > 1:          mid = math.ceil(len(*arr*)/2)          left = *arr*[:mid]          right = *arr*[mid:]          mergeSort(left)          mergeSort(right)          i = j = k = 0  *while* i < len(left) and j < len(right):  *if* left[i] < right[j]:  *arr*[k] = left[i]                  i += 1  *else*:  *arr*[k] = right[j]                  j += 1              k += 1  *while* i < len(left):  *arr*[k] = left[i]              i += 1              k += 1  *while* j < len(right):  *arr*[k] = right[j]              j += 1              k += 1 |
| Dry run | input  5 9 11  Split [5, 9, 11]  Split [5]  Merge [5]  Split [9, 11]  Split [9]  Merge [9]  Split [11]  Merge [11]  Merge [9, 11]  Merge [5, 9, 11] |
| Time Complexity | **Best:** O(n log n)  **Worst:** O(n log n |

**Proof of correctness**

A lot of proof techniques can be used, such as induction proof and contradiction proof can be used to prove the correctness but we will use "loop invariants". The loop invariant we will use is

we need to check three conditions:

* Initialization
* Maintenance
* Termination

**Strength**

* Data of any size will be sorted in efficient time.
* Faster than insertion and bubble sort as it does not have to go through all the data more than one time because it just keeps on splitting the data until it is sorted and then merge it.
* Its execution time is fixed.

**Weaknesses**

* Its execution time is slower when given data is small in size as compared to some other sorting algorithms.
* It will always go through all the data even it is sorted.
* It requires extra storage.

**Bubble Sort**

**Description**

Bubble sort consists of repeatedly stepping through a list, comparing adjacent elements, and swapping those that are out of order.

Bubble sort algorithm moves through the whole array many times to compare the elements and swap them if they are in the wrong order. The algorithm keeps on doing on the same process until it is sorted.

|  |  |
| --- | --- |
| Pseudo Code | Def BubbleSort(arr):  n = len(arr)  for i in range(0,n-1):  flag = false  for j in range(0,n-1):  if arr[j] > arr[j+1] then  swap( arr[j], arr[j+1] )  flag= true  if(!flag):  break |
| Code in Python | def BubbleSort(*arr*):  *for* i *in* range(len(*arr*)):          flag = False  *for* j *in* range(0, len(*arr*)-i-1):  *if* *arr*[j] > *arr*[j+1] :  *arr*[j], *arr*[j+1] = *arr*[j+1], *arr*[j]                  flag = True  *if* flag == False:  *break* |
| Dry run | input:  10 43 4 9 5  i=0  10 43 4 9 5  i=1  10 4 9 5 43  i=2  4 9 5 10 43  i=3  4 5 9 10 43  i=4  4 5 9 10 43 |
| Time Complexity | **Best:** O(n)  **Worst:** O(n^2) |

**Proof of correctness**

A lot of proof techniques can be used, such as induction proof and contradiction proof can be used to prove the correctness but we will use "loop invariants". The loop invariant we will use is

we need to check three conditions:

* Initialization
* Maintenance
* Termination

**Strengths**

* It is popular and easy to use.
* As the elements are swapped rather than making an extra array to sort in it so the usage of storage is low in it.
* Bubble sort is an efficient and fast for some specified data.

**Weaknesses**

* Bubble sort is an inefficient and slow when array contains a data that has a huge size.
* As it has to through all the list and swap the elements so it requires a small array to work fast.
* It can not be applied on real world algorithms.

**Quick Sort**

**Description**

Quick Sort Algorithm works on divide and conquer technique. The main working step of this algorithm is to select pivot. Pivot selection are of the following types:

* Pick First or Last element as pivot.
* Pick median as pivot.
* Pick any arbitrary element as pivot.

Partition plays an important role in Quick Sort Algorithm

|  |  |
| --- | --- |
| Pseudo Code | def quickSort(arr[], low, high)  {  if (low < high)  {  pi = partition(arr, low, high);  quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high);  }  }  def partition (arr[], low, high)  {  pivot = arr[high];  i = (low - 1)  for (j = low; j <= high- 1; j++)  {  if (arr[j] < pivot)  {  i++;  swap arr[i] and arr[j]  }  }  swap arr[i + 1] and arr[high])  return (i + 1)  } |
| Code in Python | def quickSort(*arr*, *low*, *high*):  *if* len(*arr*) == 1:  *return* *arr*  *if* *low* < *high*:          pi = partition(*arr*, *low*, *high*)          quickSort(*arr*, *low*, pi-1)          quickSort(*arr*, pi+1, *high*)  def partition(*arr*, *low*, *high*):      i = (*low*-1)      pivot = *arr*[*high*]  *for* j *in* range(*low*, *high*):  *if* *arr*[j] < pivot:              i += 1  *arr*[i], *arr*[j] = *arr*[j], *arr*[i]  *arr*[i+1], *arr*[*high*] = *arr*[*high*], *arr*[i+1]  *return* (i+1) |
| Dry run | input:  10 43 4 9 5  i=0  10 43 4 9 5  i=1  10 4 9 5 43  i=2  4 9 5 10 43  i=3  4 5 9 10 43  i=4  4 5 9 10 43 |
| Time Complexity | **Best:** O(n)  **Worst:** O(n^2) |

**Proof of correctness**

A lot of proof techniques can be used, such as induction proof and contradiction proof can be used to prove the correctness but we will use "loop invariants". The loop invariant we will use is

we need to check three conditions:

* Initialization
* Maintenance
* Termination

**Strengths**

* Quick sort is really fast in sorting means that if we run it side by side with other algorithms it will sort faster than them.
* It does not consume extra storage.
* Best sorting algorithm

**Weaknesses**

* In the worst-case scenario, quick sort has the same working performance as bubble, insertion or selections sort which makes it look slower.
* If the given list is sorted, bubble sort is much efficient than quick sort.
* If the given list is of Integers than radix sort is more efficient than quick sort.

**Counting Sort**

**Description**

Counting sort uses keys between a specific range in order to sort. Objects are sorted by counting the number of key values that are different.

|  |  |
| --- | --- |
| Pseudo Code | CountSort(input, k)    count ← array of k + 1 zeros  output ← array of same length as input    for i = 0 to length(input) - 1  j = key(input[i])  count[j] = count[j] +1  for i = 1 to k do  count[i] = count[i] + count[i - 1]  for i = length(input) - 1 downto 0  j = key(input[i])  count[j] = count[j] -1  output[count[j]] = input[i] |
| Code in Python | def CountingSort(*arr*):      max\_num = max(*arr*)      min\_num = min(*arr*)      k = max\_num - min\_num + 1      count = [0 *for* x *in* range(k)]      output = [0 *for* y *in* range(len(*arr*))]    *for* i *in* range(0, len(*arr*)):          count[*arr*[i]-min\_num] += 1  *for* i *in* range(1, len(count)):          count[i] += count[i-1]  *for* i *in* range(len(*arr*)-1, -1, -1):          output[count[*arr*[i] - min\_num] - 1] = *arr*[i]          count[*arr*[i] - min\_num] -= 1    *return* output |
| Dry run | Input  6 4 9 0 21  Output  0 4 6 9 21 |
| Time Complexity | **Best:** O (n+k)  **Worst:** O(n+k) |

**Proof of correctness**

A lot of proof techniques can be used, such as induction proof and contradiction proof can be used to prove the correctness but we will use "loop invariants". The loop invariant we will use is

we need to check three conditions:

* Initialization
* Maintenance
* Termination

**Strengths**

* It has time complexity.
* It is stable sort**.**

**Weaknesses**

* Only works on integer type of data.
* For negative integer values some changes would be required.

**Radix Sort**

**Description**

In Radix sorting, keys that share a similar significant position and value are grouped by individual digits, which then make up the keys. To sort an array of numbers, Radix sort uses counting sort as a subroutine.

|  |  |
| --- | --- |
| Pseudo Code | for i=0 to max -1  a = 10^i+1        p=10^i     for j := 0 to n-1           e= array[j] mod a           index=e / p           pocket[index].append(array[j])        count = 0  for j=0 to radix do           while pocket[j] is not empty              array[count]= get first node of pocket[j] and delete              count=count +1 |
| Code in Python | def countingSort(csvData, arr, exp):          count = [0 for x in range(10)]          # result = [0 for y in range(len(arr))]          for i in range(0, len(arr)):              index = arr[i] // exp              count[index % 10] += 1          for i in range(1, 10):              count[i] += count[i-1]          for i in range(len(arr)-1, -1, -1):              index = arr[i] // exp              csvData.row[count[index % 10] - 1] = csvData.row[i]              count[index % 10] -= 1          # for j in range(0, len(arr)):          #     arr[j] = result[j]      def RadixSort(csvData, arr):          max\_num = max(arr)          exp = 1          while max\_num / exp > 0:              sort.countingSort(csvData, arr, exp)              exp \*= 10          return csvData |
| Dry run | input:  244 153 271 141 900  Sorting by least significant digit  900 271 141 153 244  Sorting by 2nd least significant digit  900 141 244 153 271  Sorting by Most significant digit  141 153 244 271 900 |
| Time Complexity | **Best:** O(nk)  **Worst:** O(nk) |

**Proof of correctness**

A lot of proof techniques can be used, such as induction proof and contradiction proof can be used to prove the correctness but we will use "loop invariants". The loop invariant we will use is

we need to check three conditions:

* Initialization
* Maintenance
* Termination

**Strengths**

* Faster when input is smaller
* It is stable sort

**Weaknesses**

* Requires additional memory
* Slower than merge and counting sort

**Bucket Sort**

**Description**

In Bucket sort an array of elements is sorted by bucketing them into a number of buckets, after that each bucket is sorted individually

|  |  |
| --- | --- |
| Pseudo Code | Create n buckets  Initialize each bucket with 0 values  Put elements into buckets matching the range for all the buckets  Sort elements in each bucket  Gather elements from each bucket |
| Code in Python | def bucketSort(arr):      bucket = []      for i in range(len(array)):          bucket.append([])      for j in array:          index\_b = int(10 \* j)          bucket[index\_b].append(j)      for i in range(len(array)):          bucket[i] = sorted(bucket[i])      k = 0      for i in range(len(array)):          for j in range(len(bucket[i])):              array[k] = bucket[i][j]              k += 1      print(arr) |
| Dry run | Input  6 4 9 0 21  Output  0 4 6 9 21 |
| Time Complexity | **Best:** O(n+k)  **Worst:** O(n^2)) |

**Proof of correctness**

* It is proper to order items within the same bucket in the order they appear.
* Even if two items are in two different buckets, they are still in the right order.

**Strengths**

* Quicker than bubble sort
* Reduces the number of comparisons

**Weaknesses**

* Can not apply to all data types
* Depends on the number of buckets

**Tim Sort**

**Description**

Tim sort has been derived from insertion sort and merge sort which performs on data based on real world. The algorithm finds the data that is already in ordered form and then uses ordered ones to sort the remaining. So, Tim sort divide the given array into number of blocks, After making blocks sort them with the help of insertion sort and with the help of merge function merge the sorted blocks.

|  |  |
| --- | --- |
| Pseudo Code | Divide the given array into number of blocks  After making blocks sort them with the help of insertion sort  With the help of merge function merge the sorted blocks |
| Code in Python | MINIMUM= 32        def find\_minrun(n):          r = 0          while n >= sort.MINIMUM:              r |= n & 1              n >>= 1          return n + r        def insertion\_sort(array, left, right):          for i in range(left+1,right+1):              element = array[i]              j = i-1              while element<array[j] and j>=left :                  array[j+1] = array[j]                  j -= 1              array[j+1] = element          return array        def merge(array, l, m, r):            array\_length1= m - l + 1          array\_length2 = r - m          left = []          right = []          for i in range(0, array\_length1):              left.append(array[l + i])          for i in range(0, array\_length2):              right.append(array[m + 1 + i])            i=0          j=0          k=l            while j < array\_length2 and  i < array\_length1:              if left[i] <= right[j]:                  array[k] = left[i]                  i += 1                else:                  array[k] = right[j]                  j += 1                k += 1            while i < array\_length1:              array[k] = left[i]              k += 1              i += 1            while j < array\_length2:              array[k] = right[j]              k += 1              j += 1        def timSort(array):          n = len(array)          minrun = sort.find\_minrun(n)            for start in range(0, n, minrun):              end = min(start + minrun - 1, n - 1)              sort.insertion\_sort(array, start, end)            size = minrun          while size < n:                for left in range(0, n, 2 \* size):                    mid = min(n - 1, left + size - 1)                  right = min((left + 2 \* size - 1), (n - 1))                  sort.merge(array, left, mid, right)                size = 2 \* size          return array |
| Dry run | Input:  40 10 20 42 27 25 1 19  Dividing into two arrays:  40 10 20 42 and 27 25 1 19  Sorting 1st array with Insertion sort:  40 10 20 42  After Insertion()  10 20 40 42  Sorting 2nd array with Insertion sort:  27 25 1 19  After Insertion()  1 19 25 27  After merge() on these two sorted array  1 10 19 20 25 27 40 42  So the array is sorted |
| Time Complexity | **Best:** O(n)  **Worst:** O(n log n) |

**Proof of correctness**

A lot of proof techniques can be used, such as induction proof and contradiction proof can be used to prove the correctness but we will use "loop invariants". The loop invariant we will use is

we need to check three conditions:

* Initialization
* Maintenance
* Termination

**Strengths**

* It is the fastest sorting algorithm.
* Even in its worst case it performs as fast as merge sort.

**Attributes of Entity**

This desktop application will contain several columns to show the features of the mobiles phone. The user can apply different sorting algorithm on each column individually.

The names the features containing columns are as below:

* Title
* Rating
* Screen Size
* Ram
* Storage
* Price
* User Review
* URL

According to the name of the entity, it contains the following feature of phone.

**Details**

**Title:** This attribute will contain the name/title of the mobile phone.

**Rating:** This attribute will contain the rating of the phone so that the user can see the value of the phone.

**Screen Size:** This attribute will able to show the screen size of the mobile

**Ram:** This attribute will contain the RAM of phone.

**Storage:** This attribute will contain the memory storage of the mobile phone.

**Price:** This attribute will contain the price of the phone.

**User Review:** This attribute will contain the review of the phone which will be helpful for the user.

**URL:** This attribute contains the URL of website from which the data is scrapped.

**Technical Details**

|  |  |  |
| --- | --- | --- |
| **Name** | **Data Type** | **Description** |
| Title | String | Name of the phone |
| Rating | Float | Rating of phone |
| Screen size | String | Screen size of phone |
| Ram | String | Ram of phone |
| Storage | String | Storage memory of phone |
| Price | String | Price of phone |
| User Review | String | The reviews given by users |
| URL | String | From which website it is taken |

**Technology Stack**

|  |  |
| --- | --- |
| Language | Python |
| Platform | Desktop Application |
| Frontend | Python Code |
| IDEs | PyQt5 & Visual Studio |

**Use Cases**

## This the help of user interface the working is explained.

## Use Case 1:

|  |  |
| --- | --- |
| Use Case ID | U1 |
| Name | Dataset Window |
| Actor | User |
| Description | This screen starts with some data of phones within it which have been taken from a file. Different sorts of function can be applied to the given data. The user can sort the data in ascending order and descending order on every attribute individually. The user can also search for his desired phone from the given options that can also perform specific task on each column individually, the given options are such as is equal to, not equal to which can also be performed with another operator like OR, AND. The data in the table will be sorted with the help of the input given by the user.  As the sorting of the data can be done on each column individually but it can be done by nine different methods(algorithms) and each method will take different time for sorting, so an option (time taken) is given so that the user can see which method is better to sort the data. |
| GUI |  |

## Use Case 2:

|  |  |
| --- | --- |
| Use Case ID | Screen2 |
| Name | Scrapping data |
| Actor | User |
| Description | There is an option given in the dataset window for scrapping a huge amount of data with a start/stop button, by clicking on it, the data will be fetched from a file containing a huge amount of data about phones and a progress bar to show how much data is fetched, once the progress bar has finished the dataset window will be opened to show the data in table but if the user presses the start/stop button before the completion of progress bar, the fetching of the data will be stopped and you can see the unfinished fetched data by clicking on the exit button. |
| GUI |  |

**UI Component Details**

|  |  |  |
| --- | --- | --- |
| UI Component Name | Type of UI component | Purpose of UI Component/Other details |
| Start/Stop | Button | To start and stop data scrapping |
| Exit | Button | To show scraped data |
| Clear | Button | To Clear the given inputs |
| Filter | Button | To search according to the given input |
| Table | Table | To display the data |
| Combo boxes | Combo boxes | To enter data with multiple choices |
| Sort | Button | To sort the data in the table |
| Scrap more data | Button | To scrap more data |
| Exit | Button | To exit from the program |

# **Project Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Use Case Id** | **Use Case Name** | **Member Name** | **Estimated Completion Date** |
| U01 | Main Screen | Talha and Rehan | 06/11/21 |
| U02 | Next Screen | Rehan and Talha | 06/11/21 |