**Proposer Details**

| **Group Number** | 19 |
| --- | --- |
| **Registration Number of Group Members** | 2020-CS-88 |

**Proposal Details**

|  |  |
| --- | --- |
| **Project** |  |
| **Proposed Project Title** | **Mobiles and Accessories** |
| **Executive Summary** | In this project the user will be able to view the data of different mobiles phones and accessories. For this we will scrap data from some famous web sites i.e. Alibaba, daraz.pk, what mobiles etc. The user will be able to scrap the data. There are also pause and resume buttons. There is also a progress bar available which will show the data details that are scraped. The user will be able to sort the data. The filters are also available in the system. |
| **Business Case** |  |
| **Outline the business need for the project** | It will help in searching the data mobiles and accessories. This system will provide the data at one place. The user will be able to search the data of his own choice. |
| **End user of the product** | The person who wants to buy mobile phone or accessories. |
| **Motivation for Project** | This project will help us to gather data of mobiles at one place. |
| **State the level of impact expected should the project proceed and implications of not proceeding** | The user will not be able to data anything from there. This system is only available to view the data of the given web sites. |
| **Technical Details** |  |
| **Name of Entity** | Mobile phones |
| **Attributes of Entity**  **(Minimum seven attributes/rows can be increased)** | |  |  |  | | --- | --- | --- | | *Name* | *Data Type* | *Description* | | Model | String | It will store the data about the model of the mobiles and accessories. | | Display | String | It will store the data about the brand of the mobiles i.e., Samsung, IPhone etc. | | Price | String | It will store the prices of the mobiles and accessories in Pakistani rupees. | | Ram | Int | It will store the country name of the products. | | Ratings | String | It will store the ratings of the products. | | Battery | String | It will store the operating system of the product. | | Camera | String | It will store the minimum order which the user can order**.** | |  | | | |
| **Sample of Scrapping Source** |  |
| **GitHub Repository Link** | https://github.com/Shafqat-Khan/CS261F21PID19.git |
| **Sorting Algorithms** | * Merge Sort * Quick Sort * Heap Sort * Insertion Sort * Selection Sort * Bubble Sort * Counting Sort |
| |  |  | | --- | --- | | **Algorithm Name** | **Description(Each algorithm in 2-3 lines)** | | **Merge Sort** | Merge sort is based on divide and conquer rule. It firstly divide the array into two arrays and then combine the arrays. | | **Quick Sort** | It is also a divide and conquer rule. It picks a pivot from the array and divide the array around the pivot into two halves.  . | | **Bucket Sort** | In this sorting algorithm, we use max heap to arrange list of elements in Descending order and min heap to arrange list elements in Ascending order. | | **Insertion Sort** | It is a simple sorting algorithm. It takes an element from the sorted array and store the element into the new array on the right position. | | **Selection Sort** | This algorithm repeatedly find a minimum element from the unsorted array and place the minimum element at the start of the array in the sorted manner | | **Bubble Sort** | It traverse through the list repeatedly and compare the elements and if the elements are on the wrong place it swaps the elements. | | **Counting Sort** | It sort the array on the basis of the keys between the specified ranges. And find the position of each element in the output through different arithmetic operations. | | |
| **Searching Algorithms** | Sequential Search: It is also known as linear search. It traverse through the list to find an element. It traverse through the list until the match is found or the whole list is searched. |
| **Searching Filters for each data type** | * Descending Order * Ascending Order * Starts with * Ends with * Contains * Is Empty |
| **Multi-Level Sorting** | No multi leveling sorting |
| ***Interfaces for your project*** |  |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | **UI Component Name** | **Type of UI component** | **Purpose of UI Component/Other details** | | Push Button | Load | It will load data from the csv file to the table. | | Push Button | Scrap | Button to scrap data from the given web sites link. | | Progress Bar | Progress bar | It will show the progress of the data which is scraped. | | Push Button | Pause | It will pause the scraping. | | Push Button | Resume | It will resume the scraping. | | Combo Box | Sorting Algorithms | It will show the list of algorithms that are available. | | Push Button | Sorting | It will sort the data in the selected way. | | Combo Box | Searching Algorithms | It will show the list of filters that are available. | | Scroll Bar | Scroll | It will use to scroll to view the data. | |  |  |  | | --- | --- | | **Bubble Sort** | | | **Description:** | It traverse through the list repeatedly and compare the elements and if the elements are on the wrong place it swaps the elements. | | **Code:** | def bubbleSort(arr):  size = len(arr)  for i in range(size):  for j in range(size-i-1):  if arr[j] > arr[j+1]:  arr[j],arr[j+1] = arr[j+1], arr[j] | | **Time Complexity:** | Best case : O(N)  Average case : O(N2)  Worst case : O(N2) | | **Strengths:** | * The code is small * Easy to understand * It uses less memory | | **Weakness:** | * It is one of the slowest algorithms for large amount of data. * It is a recursive algorithm. | | **Merge Sort** | | | **Description:** | Merge sort is based on divide and conquer rule. It firstly divide the array into two arrays and then combine the arrays. | | **Code:** | def mergeSort(arr):  if len(arr) > 1:  #Mid of array  mid = len(arr)//2    #Dividing the array into two halves  leftArr = arr[:mid]  rightArr = arr[mid:]  # Sorting the both halves of array seperatily  mergeSort(leftArr)  mergeSort(rightArr)  i, j, k = 0, 0, 0  while i < len(leftArr) and j < len(rightArr):  if leftArr[i] < rightArr[j]:  arr[k] = leftArr[i]  i += 1  else:  arr[k] = rightArr[j]  j += 1  k += 1  # Checking whether any element is left  while i < len(leftArr):  arr[k] = leftArr[i]  i += 1  k += 1  while j < len(rightArr):  arr[k] = rightArr[j]  j += 1  k += 1 | | **Time Complexity:** | Best case : Ω(n log(n))  Average case : Ꝋ(n log n)  Worst case : O(n log n) | | **Strengths:** | * It is quicker for large problems than bubble sort. * It has a similar time running. | | **Weakness:** | * It is slower for the small problems because it takes the same time as that of large problems. * It takes more space to store the elements in the memory. | | **Quick Sort** | | | **Description:** | It is also a divide and conquer rule. It picks a pivot from the array and divide the array around the pivot into two halves.  . | | **Code:** | def partition(low, high, arr):  index = low  pivot = arr[index]    while low < high:  while low < len(arr) and arr[low] <= pivot:  low += 1    while arr[high] > pivot:  high -= 1    if(low < high):  arr[low],arr[high] = arr[high],arr[low]  arr[high],arr[index] = arr[index],arr[high]  return high    def quickSort(low, high, array):  if (low < high):  part = partition(low, high, array)  quickSort(low, part - 1, array)  quickSort(part + 1, high, array) | | **Time Complexity:** | Best case : Ω(n log n)  Average case : Ꝋ(n log n)  Worst case : O(N2) | | **Strengths:** | * It works faster than merge sort in case of smaller problems. * It has small inner loops. * It requires less space in the memory | | **Weakness:** | * It is a recursive algorithm. * A simple mistake can lead to a worst result. | | **Insertion Sort** | | | **Description:** | It is a simple sorting algorithm. It takes an element from the sorted array and store the element into the new array on the right position. | | **Code:** | def insertionSort(arr):  for i in range(1, len(arr)):  ele = arr[i]  j = i-1  while j >= 0 and ele < arr[j]:  arr[j + 1] = arr[j]  j -= 1  arr[j + 1] = ele | | **Time Complexity:** | Best case : Ω(N)  Average case : Ꝋ(N2)  Worst case : O(N2) | | **Strengths:** | * It requires constant amount of small memory. * It is a simple algorithm. * It is efficient for small problems. | | **Weakness:** | * It requires a more time to sort big problems as compare to quick sort. * The insertion sort is useful only when sorting list contains few items.. | | **Selection Sort** | | | **Description:** | This algorithm repeatedly find a minimum element from the unsorted array and place the minimum element at the start of the array in the sorted manner | | **Code:** | def selectSort(arr):  for i in range(len(arr)):  minIdx = i  for j in range(i+1, len(arr)):  if arr[minIdx] > arr[j]:  minIdx = j  arr[i],arr[minIdx] = arr[minIdx],arr[i]  arr = [2, 3, 54, 6, 9, 5]  selectSort(arr)  print(arr) | | **Time Complexity:** | Best case : Ω(N)  Average case : Ꝋ(N2)  Worst case : O(N2) | | **Strengths:** | * It does nor requires additional space for storing the elements. * It performs well on small number of problems. | | **Weakness:** | * It is poor efficient in dealing with larger problems. * It is less efficient than quick sort. | | **Counting Sort** | | | **Description:** | It sort the array on the basis of the keys between the specified ranges. And find the position of each element in the output through different arithmetic operations. | | **Code:** | def countingSort(arr):  size = len(arr)  arr1 = [0] \* size  count = [0] \* 10  for i in range(0, size):  count[arr[i]] += 1  for i in range(1, 10):  count[i] += count[i-1]  i = size-1  while i >= 0:  arr1[count[arr[i]]-1] = arr[i]  count[arr[i]] -= 1  i -= 1  for i in range(0, size):  arr[i] = arr1[i] | | **Time Complexity:** | Best case : Ω(n + k)  Average case : Ꝋ(n + k)  Worst case : O(n + k) | | **Strengths:** | * It requires linear time to sort the array. * It requires less space in the memory. | | **Weakness:** | * In case of negative numbers the complexity increases. * It work for discrete integers only. | | **Bucket Sort** | | | **Description:** | It divide the unsorted array into different buckets and then sort the buckets individually by using one of the suitable sorting algorithm. | | **Code:** | def bucketSort(arr):  bucket = []  for i in range(len(arr)):  bucket.append([])  for j in arr:  index = int(10 \* j)  bucket[index].append(j)  for i in range(len(arr)):  bucket[i] = sorted(bucket[i])  k = 0  for i in range(len(arr)):  for j in range(len(bucket[i])):  arr[k] = bucket[i][j]  k += 1 | | **Time Complexity:** | Best case : Ω(n + k)  Average case : Ꝋ(n + k)  Worst case : O(n2) | | **Strengths:** | * It is quicker than bubble sort. * It reduces the number of comparisons. | | **Weakness:** | * It cannot be applied on all kind of data types. * It requires some extra space to make the buckets | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | | |