| Group Number | 37 |
| --- | --- |
| Registration Number of Group Members | 1. 2020-CS-68 2. 2020-CS-73 |

|  |  |
| --- | --- |
| ***Project*** |  |
| Project Title | Apple Products |
| Executive Summary | Firstly, on the top, there will be given a textbox in which user can start the url of the specific category of apple website whose data he wants to scrap.  After that, there will be mentioned some categories of the products. User can select any category depends upon his need or taste. User will select any button he wants to, a new page will open which will contain the informtion of the specific category.  User can sort that data with an algorithm of his choice. For this, there will be a list of algorithms given, user can select any one from all of them. He will be able to select that whether he wants to sort data in ascending or desending order. User will also have an option to search data provided in the filters of each column heading. |

|  |  |
| --- | --- |
| ***Project Link*** |  |
| GitHub Repository Link | https://github.com/ridsa00/CS261F21PID37 |

|  |  |
| --- | --- |
| ***Technical Details*** |  |
| Name of Entity | * MacBook * iPhone |
| Attributes | |  |  |  | | --- | --- | --- | | *Name* | *Data Type* | *Description* | | Model | String | This will tell us about the model of chosen product. | | Color | String | This will tell us about the color of chosen product. | | Price | Integer | This will tell us about the price of chosen product. | | Rating | Double | This will tell us about the rating of chosen product. | | Design | String | This will tell us about the design of chosen product. | | Storage | Integer | This will tell us about the storage of chosen product. | | CPU | Integer | This will tell us about the power of CPU. | | RAM | String | This will tell us about the RAM of chosen product. | |

|  |  |
| --- | --- |
| ***Name*** | ***Sample UI in Pencil Tool*** |
| Login form | Firstly we were about to scrap data of all of these categories but due to the lack of availability of data, we just implement two categories which are MacBook and iPhones given below in detail as well. |
| Laptop form |  |
| iPhone form |  |

|  |  |
| --- | --- |
| ***Data Scrapping*** |  |
| Sites | MacBook:  <https://www.ebay.com/sch/i.html?_from=R40&_trksid=p2540003.m570.l1313&_nkw=macbook&_sacat=0>  iPhone:  https://www.ebay.com/sch/i.html?\_nkw=iphone&\_sop=12&\_pgn=0 |
| URLs | iPhone:  <https://drive.google.com/file/d/1hbI6TWSI72raRnXhANVQMi2MofesKndW/view?usp=sharing>  MacBook:  <https://docs.google.com/spreadsheets/d/1A7z0pG7RDJwyRSDjRQOpe_u8pRV9xhO5/edit?usp=sharing&ouid=104574891620464810503&rtpof=true&sd=true>  Products:  https://drive.google.com/file/d/1g4Ifr3tuSKx7rZgH4f-Qh9XW3S4PmJeV/view?usp=sharing |
| Hurdles | Our mission was to scrap data from apple website. Unfortunately, we were unable to scrap data from their official website due to their privacy. Then we tried to scrap data from different websites like amazon, daraz, myntra etc. on these websites, data was limited and we were still unable to get 1 million data. Now we came to know a new website name as ebay, we scrapped data of MacBook and iPhone from this website by the above providin |

|  |  |
| --- | --- |
| ***Sorting Algorithms*** |  |
| **Insertion Sort** | In insertion sort, we considered the first element of the array as sorted, even the array is unsorted. In insertion sort each element is compared with the previous element of the array, resulting growing sorted output list.  With each iteration, sorting algorithm removes one element at a time from the array and inserts the sorted element in the appropriate location in sorted array. This process continues until the whole array is sorted.  Insertion sort is very simple sorting algorithm. It is efficient for small data list. It can sort the list as it receives data. For the whole operation it needs only a constant amount of memory space. This sorting algorithm is not efficient for large data list.  Insertion sort is more efficient than bubble sort or selection sort and less efficient than heap sort and quick sort algorithms. |
| **Merge Sort** | Merge sort is the sorting algorithm commonly used in computer science. Merge sort is basically known as divide and conquer algorithm. In merge sort, we recursively breaking down the array into two sub arrays of same related types.  Now the arrays become easy to sort. The solutions applied on both sub arrays and then combine to give actual solution to the unsorted array. Merge sort is very useful for sorting the linked list. Merge sort is very stable sort which means every element in array maintain their original position with respect to each other.  Merge sort is very efficient in worst case. The time complexity for the merge sort is O (n log n). |
| **Quick Sort** | If we want to sort the array quickly then we use quick sort  algorithm. Quick sort is also known as divide and conquer. In the quick sort we select one element from the array as pivot and then we start sorting the array by moving smaller numbers from the left side of the pivot and greater numbers in the array to the right side of the pivot element.  This process is done for each sub array of smaller values as well as for the sub array with greater values. It is fastest sorting algorithm. This fastest algorithm is used for the searching process.  Quick sort is constant faster than the merge sort. Quick sort is very useful for sorting arrays. In this sorting algorithm the relative order of equal sort items is not preserved so, in efficient implementation quick sort is not stable sort. The time complexity for quick sort is O (n log n). |
| **Bubble Sort** | In bubble sort, we comparing each pair of adjacent items and swapping them if they are at wrong position. This process repeatedly done until no swaps are required which show that list is sorted. Bubble sort is also known as sinking sort or comparison sort.  The position of elements in the list plays an important role in the bubble sort, the small elements in the end move toward the start very slowly but large elements easily swap to their real position. That’s why these elements are called rabbits and turtles.  In quick sort, after every pass, all elements after the last swap are sorted so, there is no need to check again, thereby skipping the tracing of variables.  Bubble sort has a worst case and average complexity of O (n2). Used to identify the list is already sorted or not. If sorted (which is the best case) the complexity of bubble sort is only O (n). |
| **Selection Sort** | Selection sort is selecting the smallest element from the remaining elements of the list and then replace it with element which is at its correct position.  Selection sort is suitable for small data set like an insertion sort. In this sort we find the least element and move it accordingly. This sort is simpler than insertion sort and less efficient than insertion sort.  This sort performs well on already sorted items in the list. This sort does not require a lot of space, only extra space required to store temporal variables. This sort is very fast for some specific applications. |
| **Bucket Sort** | Bucket sort is the sorting algorithm in which data items are distributed in the buckets equally and these buckets hold the same type of data. Then on each bucket different algorithm apply and then merge all to form a main sorted list.  The time complexity of the bucket sort for best case is O (n + k) and for worst case it is O (n^2). For worst case, space complexity is O (nk). Bucket sort is used when the data is uniformly distributed over the range.  This sorting algorithm is faster than a bubble sort. Bucket sort goes from MSD to LSD. If we use highly clustered values then it’s not worth it. Bucket sort is can’t apply on all data types. |
| **Sleep Sort** | In sleep sort, we start a separate task for each element of array in order to sort it. Each task sleeps for a specific interval according to a key that is assigning to it. |
| **Counting Sort** | In counting sort, we sort the elements by technique of counting numbers of each unique element in the array. In this sorting algorithm the elements are sorted in such a way that each element is greater than its previous element and lesser than to the next element.  Counting sort is not stable sorting algorithm because the relative order of the elements in the sorted array is not preserved in the final sorted array. If we want stability then counting sort is not a best option.  Counting sort is only used for integers. This sorting algorithm has lesser time complexity. Time complexity of counting sort in best case is O (n + k) and worst case it is O (n). |

|  |  |
| --- | --- |
| ***Algorithm Name*** | ***Pseudo Code*** |
| **Insertion Sort** | **for** j 🡨 2 **to** length[A]  **do** key 🡨 A[j]  insert A[j] into sorted  sequence A [1...j-1]  i🡨j-1  **while** i > 0 or A[i]>key  **do** A[i+1] 🡨 A[i]  i🡨i-1  A[i+1]🡨 key |
| **Merge Sort** | **Merge** (A, p, q, r)  n1 = q-p+1  n2 = r-q  let L [1, …, n1+1] and R = [1, …, n2+1] be two new arrays  **for** i = 1 to n1  L[i] = R[p-i+1]  **for** j =1 to n2  R[j] = A[q+j]  L[n1+1] = ~  R [n2+1] = ~  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, q+1, r)  **Merge** (A, p, q, r) |
| **Quick Sort** | Quicksort (A, p, r)  **If** p<r Partition (A, p, r)  q = Partition (A, p, r) a = A[r]  Quicksort (A, p, q-1) i = p-1  Quicksort (A, q+1, r) **for** j =p **to** r-1  **if** A[j] <= a  i = i+1  exchange A[i] with A[j]  exchange A[i+1] with A[j]  **return** i+1 |
| **Bubble Sort** | **Bubble-Sort** (arr A)  n 🡨 length of A  **for** I 🡨 n-1  **do** **for** j 🡨 0 **to** n-i  **do if** A[j] < A[j+1]  then swap A[j] 🡨 A[j+1] |
| **Selection Sort** | **Selectionsort** (A, n)  **for** i=1 **to** n-1 do  min 🡨 i  **for** j=i+1 **to** n do  **if** A[j] < A[min] then  min 🡨j  swap A[i], A[min]  end |
| **Bucket Sort** | **Bucket-Sort**(A)  n🡨 length[A]  **for** i🡨1 to n  **do** insert A[i] into list B[A[i]]  **for** i🡨0 **to** n-1  **do** sort list B[i] using insertion sort  concatenate the lists B [1], B [2], …. B[n-1] together in order |
| **Sleep Sort** | **Procedure number**(n)  Sleep n seconds  Print(n)  End  **For** arg in args  Run number(arg) in background  End  Wait for all processes to finish |
| **Counting Sort** | Countingsort (A, B, n, k)  **for** i🡨0 **to** k  **do** C[i] 🡨 0  **for** j🡨1 **to** n  **do** C[A[j]] 🡨C[A[j]] +1  **for** i🡨 1 **to** k  **do** C[i] 🡨 C[i] + C[i-1]  **for** j🡨 n **downto** 1  **do** B[C[A[j]]] 🡨 A[j]  C[A[j]] 🡨 C[A[j]] -1 |

|  |  |
| --- | --- |
| ***Sorting Algorithm*** | ***Python Code*** |
| **Insertion Sort** | def insertionSort(array):  for i in range (1, len(array)):  key = array[i]  j = i-1  while j >= 0 and key < array[j]:  array [j + 1] = array[j]  j -= 1  array [j + 1] = key  array = [23, 11, 73, 15, 6]  insertionSort(array)  for i in range(len(array)):  print ("% d" % array[i]) |
| **Merge Sort** | def mergeSort(array):  if len(array) > 1:  r = len(array)//2  L = array [: r]  M = array[r:]  mergeSort(L)  mergeSort(M)  i = j = k = 0  while i < len(L) and j < len(M):  if L[i] < M[j]:  array[k] = L[i]  i += 1  else:  array[k] = M[j]  j += 1  k += 1  while i < len(L):  array[k] = L[i]  i += 1  k += 1  while j < len(M):  array[k] = M[j]  j += 1  k += 1  def printList(array):  for i in range(len(array)):  print(array[i], end=" ")  print ()  if \_\_name\_\_ == '\_\_main\_\_':  array = [6, 5, 12, 10, 9, 1]  mergeSort(array)  print ("Sorted array is: ")  printList(array) |
| **Quick Sort** | def partition (start, end, array):  pivot\_index = start  pivot = array[pivot\_index]  while start < end:  while start < len(array) and array[start] <= pivot:  start += 1  while array[end] > pivot:  end -= 1  if (start < end):  array[start], array[end] = array[end], array[start]  array[end], array[pivot\_index] = array[pivot\_index], array[end]  return end  def quick\_sort (start, end, array):  if (start < end):  p = partition (start, end, array)  quick\_sort (start, p - 1, array)  quick\_sort (p + 1, end, array)  array = [ 10, 7, 8, 9, 1, 5]  quick\_sort 0, len(array) - 1, array)  print (f'Sorted array: {array}') |
| **Bubble Sort** | def bubbleSort(array):  for i in range(len(array)):  for j in range (0, len(array) - i - 1):  if array[j] > array [j + 1]:  temp = array[j]  array[j] = array[j+1]  array[j+1] = temp  data = [-2, 45, 0, 11, -9]  bubbleSort(data)  print ('Sorted Array in Ascending Order:')  print(data) |
| **Selection Sort** | def selectionSort (array, size):  for step in range(size):  min\_idx = step  for i in range (step + 1, size):  if array[i] < array[min\_idx]:  min\_idx = i  (array [step], array[min\_idx]) = (array[min\_idx], array[step])  data = [-1, 45, 0, 21, -9]  size = len(data)  selectionSort (data, size)  print ('Sorted Array:')  print(data) |
| **Bucket Sort** | def bucketSort(array):  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  return array  array = [.42, .32, .33, .52, .37, .47, .51]  print("Sorted Array in descending order is")  print(bucketSort(array)) |
| **Sleep Sort** | from time import sleep  from threading import Timer    def sleepsort(values):  sleepsort. result = []  def add1(x):  sleepsort. result. append(x)  mx = values [0]  for v in values:  if mx < v: mx = v  Timer (v, add1, [v]). start ()  sleep(mx+1)  return sleepsort. result    numbers = [2,4,6,8,10,9,7,5,3,1]  print(numbers)    numbers = sleepsort(numbers)  print(numbers) |
| **Counting Sort** | def countingSort (array):  size = len(array)  output = [0] \* size  count = [0] \* 10  for i in range (0, size):  count[array[i]] += 1  for i in range (1, 10):  count[i] += count [i - 1]  i = size - 1  while i >= 0:  output[count[array[i]] - 1] = array[i]  count[array[i]] -= 1  i -= 1  for i in range (0, size):  array[i] = output[i]  data = [4, 6, 2, 1, 0, 3, 1]  countingSort(data)  print ("Sorted Array: ")  print(data) |

|  |  |  |
| --- | --- | --- |
| ***Algorithm*** | ***Strength*** | ***Weakness*** |
| **Insertion Sort** | **🡪** The main advantage of the insertion sort is its simplicity.  **🡪** When it is dealing with small list, it exhibits good performance.  **🡪** Relative order of the items  with equal keys does not change. | **🡪** It is not efficient to deal with large list.  **🡪** The major disadvantage is its average time complexity of O(n^2).  **🡪** Does not perform as well as others**.** |
| **Merge Sort** | **🡪**It can sort large list’s inputs quickly and efficiently. Unlike insertion and bubble sort go through the whole list.  **🡪**It is more stable than the other algorithms.  **🡪**It starts with smaller sub files and finishes with largest one so it doesn’t need stack. | **🡪**It is slower than the other sorting algorithms.  **🡪**It uses some more additional memory space to store the elements. |
| **Quick Sort** | **🡪** Requires only n (log n) time to sort n elements.  **🡪** Has an extremely short inner loop.  **🡪** In case of smaller array size of datasets | **🡪** It is recursive.  **🡪** Requires quadratic time in worst case.  **🡪** It is fragile. |
| **Bubble Sort** | **🡪** It is simple to write and easy to understand.  **🡪** It need only few lines of codes and needs less space.  **🡪** It is an efficient way to check either a list is already in sorted order. | **🡪** The major disadvantage is the amount of time it takes to sort.  **🡪** It is pretty slow.  **🡪** It doesn’t deal well with a list containing a huge number of items. |
| **Selection Sort** | **🡪** It does not depend upon initial arrangement of data.  **🡪** Does not require a lot of space for sorting.  **🡪** Perform well on small list. | **🡪** It is not efficient with large number of inputs.  **🡪** it is not scalable  **🡪** It requires n-square number of steps for sorting n elements. |
| **Bucket Sort** | **🡪** One of the main advantages is that it is quicker to run than other algorithms.  **🡪** Putting data into small buckets in order to sort them individually can reduce the number of comparisons that need to be carried out. | **🡪** Biggest problem is that it is a bit complicated to describe it for computer.  **🡪** Other than the numbers, it cannot sort names or other strings. |
| **Sleep Sort** | **🡪**It can perform efficiently as it’s time complexity is O (n\*logn). | **🡪** This algorithm’s won’t work for negative numbers as a thread cannot sleep for a negative amount of time.  **🡪** You may not get the correct output all the time. |
| **Counting Sort** | **🡪** Has lesser time complexity.  **🡪** Not based on differences.  **🡪** Running time is linear. | **🡪** Can only be used for integers elements**.**  **🡪**Restricted Inputs.  **🡪** Space cost. |

|  |  |
| --- | --- |
| ***Searching Algorithm*** |  |
| **Binary Search** | In binary search, the element to be searched is compare with the middle element of the array. If we get match then we return the index of the middle element. If we don’t get a match then we check the element to be searched is less or greater than the middle element.  If the element to be searched is greater than middle element in value than we pick element to the right side of the array.  If the element to be searched is lesser than the middle element than we pick this element to the left side of the array.  Binary searching has simple implementation and it is great to search through large sorted arrays. It has best time complexity O (log n). |
| **Linear Search** | Linear search is very simple searching algorithm. In this algorithm, we search element in the list by traversing the array from the start, till the desired element is found.  In this algorithm, we compare the element to be searched with other elements and if matching element is found then it returns the index of this element otherwise it returns -1.  It has simple implementation and has time complexity of O(n) which means the time is linearly dependent on the number of elements, which is not bad but not that goof too. It is used for unsorted or unordered small list of elements. |

|  |  |
| --- | --- |
| ***Algorithm Name*** | ***Pseudo Code*** |
| **Binary Search** | Procedure binary\_search  A🡨sorted array  n🡨size of array  x🡨value to be searched  Set lowerBound = 1  Set upperBound = n  while x is not found  if upperBound < lowerBound  x does not exist  set midpoint = lowerBound + (upperBound + lowerBound) / 2    if A[midpoint] < x  set lowerBound = midpoint +1  if A[midpoint] >x  set upperBound = midpoint – 1  if A[midpoint] = x  x found at location midpoint  end while  end Procedure. |
| **Linear Search** | Procedure linear\_search (list, value)  for each item in the mist  if match item == value  return the items location  end if  end for  end procedure |

|  |  |
| --- | --- |
| ***Searching Algorithm*** | ***Python Code*** |
| **Binary Search** | def binarySearch (array, x, low, high):  while low <= high:  mid = low + (high - low)//2  if array[mid] == x:  return mid  elif array[mid] < x:  low = mid + 1  else:  high = mid - 1  return -1  array = [3, 4, 5, 6, 7, 8, 9]  x = 4  result = binarySearch (array, x, 0, len(array)-1)  if result! = -1:  print ("Element is present at index " + str(result))  else:  print ("Not found") |
| **Linear Search** | def linearSearch (array, n, x):  for i in range (0, n):  if (array[i] == x):  return i  return -1  array = [2, 4, 0, 1, 9]  x = 1  n = len(array)  result = linearSearch (array, n, x)  if (result == -1):  print ("Element not found")  else:  print ("Element found at index: ", result) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***UI Components*** | ***Quantity*** | | | | |
| Forms | Push buttons | Label | Line edit | Table widget | Combo box |
| Login form | 2 | 3 | 0 | 0 | 0 |
| Laptop form | 2 | 3 | 1 | 1 | 2 |
| iPhone form | 3 | 3 | 0 | 1 | 2 |

|  |  |
| --- | --- |
| ***User Interface (UI)*** |  |
| Login form |  |
| Laptop form |  |
| iPhone form |  |

|  |  |
| --- | --- |
| ***Tasks*** | ***Planning and collaboration*** |
| Scrapping | Due to a lot of hurdles, we were not able to meet the deadline of scrapping and we did it after deadline. |
| Sorting algorithms | That word document on sorting algorithm was divided among both of us. Half of the algorithms were done by the one partner and the remaining half of them was done by the other partner.  Due to the net issue in hostel, we could not submit that work on time and done in late submission. |
| Integration | This part was quite hard to implement, as we did not scrap data till yet due to a lot of issues. When we understand this part of the project, then we both start working on this part according to our understanding. |
| Report | We did the report according to the tasks divided. |
| ***Note*** | There were no targeted deadlines for our project. The deadline which was given to us by the instructor were considered during the whole project and we just work on that part which was assigned according to the deadline. |