Scrapping & Sorting

Session: 2022 - 2026

Group ID
DSAMidProjectPID25

Submitted By:

Ch.Noman Ahmad	2022-CS-91
Fasi-ur-Rehman	2022-CS-71

Supervised By:

Sir Nazeef Ul Haq

Department of Computer Science
University of Engineering and Technology
Lahore Pakistan

Contents

1 Summary

1.1 Description

The purpose of this application is provide all the insights of apps and games. For the purpose of collecting the data we choose one of the most famous apk website which is “Apk Pure”. On this app not only android apps but IOS apps are also present. That’s why we can cover the wide range of applications in this application.

his application will provide user with user friendly interface. It will present all the attributes of all the applications on the website in a visually attractive way. It will have option for the user to sort the applications on the base of different things. In this way he can select best app on the basis of rating in a certain category.

This app will have a login system. So that if you login as admin you can change the things in short you will have the ability to perform CRUD. If you login as a user you can only view the data or search it or sort it. You cannot change the data as a user.

1.2 End User

The end user can anyone and he/she may use it for learning purposes, data visualization, learning how sorting or searching algorithms work in our daily life and data analysis.

2 Algorithms and Operations

2.1 Sorting Algorithms

2.1.1 Bubble Sort

In this algorithm we compare the elements of adjacent indexes and the one which is greater will be shifted to the right and this way largest element will be placed at right most index. In this way array is sorted in this algorithm.

Pseudo Code

```
Procedure BubbleSort(A: array of items, n: integer)
  for i = 0 to n − 1
    for j = 0 to n − i − 1
      if A[j] > A[j + 1]
        swap A[j] and A[j + 1]
  end for
end for
```

2.1.2 Selection Sort

In this algorithm we first find the minimum number in an unsorted array and then we place it to left most index of unsorted array. Then we say left part is sorted then we find the minimum in unsorted array and place it to the left of array. In this way array will be sorted.

Pseudo Code

```
Procedure SelectionSort(A: array of items, n: integer)
  for i = 0 to n − 1
    minIndex = i
    for j = i + 1 to n
      if A[j] < A[minIndex]
        minIndex = j
    end for
    swap A[i] and A[minIndex]
end for
```

2.1.3 Insertion Sort

Insertion sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part.

Pseudo Code

```
Procedure InsertionSort(A: array of items, n: integer)
  for i = 1 to n − 1
    key = A[i]
    j = i − 1
    while j ≥ 0 and A[j] > key
      A[j + 1] = A[j]
      j = j − 1
    A[j + 1] = key
end for
```

2.1.4 Quick Sort

In this algorithm we follow divide and conquer technique. In this we select an element as the pivot and then we distribute elements on either side of pivot. The distribution is done by comparing each element with the pivot. The number which is smaller than pivot is placed at right and the number greater than pivot is placed on right. Then the same process happen for left and right part and at the end array is sorted.

Pseudo Code

```
Procedure QuickSort(A: array of items, low: integer, high: integer)
  if low < high
    pivotIndex = Partition(A, low, high)
    QuickSort(A, low, pivotIndex − 1)
    QuickSort(A, pivotIndex + 1, high)

Procedure Partition(A: array of items, low: integer, high: integer): integer
  pivot = A[low]
  left = low + 1
  right = high
  done = false
  while not done
    while left ≤ right and A[left] ≤ pivot
      left = left + 1
    while A[right] ≥ pivot and right ≥ left
      right = right − 1
    if right < left
      done = true
    else
      swap A[left] and A[right]
  end while
  swap A[low] and A[right]
  return right
```

2.1.5 Merge Sort

This algorithm is based on Divide and Conquer technique. In this we divide the array until every array reach unit length. Then we combine the units to form the array again. While combining units of array we compare the elements and then we combine them in sorted form. In this way array is sorted.

Pseudo Code

```
Procedure MergeSort(A: array of items)
  if length(A) ≤ 1
    return A
  end if
  // Split the array into two halves
  mid = length(A)/2
  left = MergeSort(A[0tomid − 1])
  right = MergeSort(A[midtoend])
  // Merge the sorted halves
  return Merge(left, right)

Procedure Merge(left: array of items, right: array of items)
  result = emptyarray
  i = 0
  j = 0
  // Compare elements in the left and right arrays
  while i < length(left)andj < length(right)
    if left[i] ≤ right[j]
      add left[i] to result
      i = i + 1
    else
      add right[j] to result
      j = j + 1
    end if
  // Add remaining elements from both arrays
  add remaining elements from left to result
  add remaining elements from right to result
  return result
```

2.1.6 Heap Sort

Heap sort is a comparison-based sorting technique based on the Binary Heap data structure. In this we find the maximum element and place it to right and we keep doing this. In this way array is sorted.

Pseudo Code

```
Procedure HeapSort(A: array of items)
  BuildHeap(A)
  heapSize = length(A)
  for i from heapSize − 1 to 1
    swap A[0] and A[i]
    heapSize = heapSize − 1
    Heapify(A, 0, heapSize)

Procedure BuildHeap(A: array of items)
  heapSize = length(A)
  for i from ⌊heapSize/2⌋ − 1 to 0
    Heapify(A, i, heapSize)

Procedure Heapify(A: array of items, i: integer, heapSize: integer)
  largest = i
  leftChild = 2i + 1
  rightChild = 2i + 2
  if leftChild < heapSize and A[leftChild] > A[largest]
    largest = leftChild
  if rightChild < heapSize and A[rightChild] > A[largest]
    largest = rightChild
  if largest ≠ i
    swap A[i] and A[largest]
    Heapify(A, largest, heapSize)
```

2.1.7 Radix Sort

Radix Sort is a non-comparative sorting algorithm that works by distributing elements into buckets based on their individual digits or radix (from right to left) and then collecting the elements from the buckets in a specific order.

Pseudo Code

```
Procedure RadixSort(A: array of non-negative integers)
  maxValue = max(A)
  exp = 1
  while ⌊maxValue / exp⌋ > 0
    CountingSort(A, exp)
    exp = exp × 10

Procedure CountingSort(A: array of non-negative integers, exp: integer)
  output: array of same size as A
  count = arrayofsize10
  for i from 0 to 9
    count[i] = 0
  for i from 0 to length(A) − 1
    index = ⌊A[i] / exp⌋ 10
    count[index] = count[index] + 1
  for i from 1 to 9
    count[i] = count[i] + count[i − 1]
  for i from length(A) − 1 to 0
    index = ⌊A[i] / exp⌋ 10
    output[count[index] − 1] = A[i]
    count[index] = count[index] − 1
  for i from 0 to length(A) − 1
    A[i] = output[i]
```

2.1.8 PigeonHole Sort

Pigeonhole Sort is a variation of Counting Sort that sorts integers within a specified range. It counts the occurrences of each integer and uses this information to sort the data.

Pseudo Code

```
Procedure PigeonholeSort(A: array of non-negative integers)
  maxValue = max(A)
  pigeonholes = arrayofsize(maxValue + 1)
  for i from 0 to maxValue
    pigeonholes[i] = emptylist
  for i from 0 to length(A) − 1
    pigeonholes[A[i]].append(A[i])
  index = 0
  for i from 0 to maxValue
    for each element in pigeonholes[i]
      A[index] = element
      index = index + 1
```

2.1.9 Hybrid Merge Sort

This algorithm is a modification of the Merge sort. In this, we combine Merge sort and Insertion sort. We use the ability of Merge sort to sort large arrays and insertion sort's ability to sort small arrays. In this, we divide the array to an optimum number n and then then we sort n numbers array using insertion sort and then we combine them using merge sort. In this way, array is sorted.

Pseudo Code

Procedure HybridMergeSort(A: array of non-negative integers)
  const $MIN\_RUN = 32$
  for i **from** 0 **to** $length(A) - 1$ **step** MIN_RUN
    InsertionSort$(A, i, \min(i + MIN\_RUN - 1, length(A) - 1))$
  $size = MIN\_RUN$
  while $size < length(A)$
    for i **from** 0 **to** $length(A) - 1$ **step** $2 \times size$
      $left = A[i : i + size]$
      $right = A[i + size : \min(i + 2 \times size, length(A))]$
      $merged = Merge(left, right)$
      $A[i : i + 2 \times size] = merged$
  $size = 2 \times size$

2.1.10 Counting Sort

Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array.

Pseudo Code

Procedure CountingSort(A: array of non-negative integers)
  $maxValue = max(A)$
  $count = arrayofsize(maxValue + 1)$
  for i **from** 0 **to** $maxValue$
    $count[i] = 0$
  for i **from** 0 **to** $length(A) - 1$
    $count[A[i]] = count[A[i]] + 1$
  output: array of same size as A
  $index = 0$
  for i **from** 0 **to** $maxValue$
    while $count[i] > 0$
      $A[index] = i$
      $index = index + 1$
      $count[i] = count[i] - 1$

2.1.11 Shell Sort

Shell sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ShellSort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every hth element are sorted.

Pseudo Code

```
Procedure ShellSort(A: array of non-negative integers)
  gap = length(A)/2
  while gap > 0
    for i from gap to length(A) − 1
      temp = A[i]
      j = i
      while j ≥ gap and A[j − gap] > temp
        A[j] = A[j − gap]
        j = j − gap
      A[j] = temp
    gap = gap/2
```

2.1.12 Cocktail Shaker Sort

Cocktail Sort is a variation of Bubble sort. It is also known as **Bi-Directional Bubble Sort** The Bubble sort algorithm always traverses elements from left and moves the largest element to its correct position in the first iteration and second-largest in the second iteration and so on. Cocktail Sort traverses through a given array in both directions alternatively. Cocktail sort does not go through the unnecessary iteration making it efficient for large arrays.

Pseudo Code

```
Procedure CocktailShakerSort(A: array of non-negative integers)
  left = 0
  right = length(A) − 1
  while left ≤ right
    for i from left to right − 1
      if A[i] > A[i + 1]
        swap A[i] and A[i + 1]
    right = right − 1
    for i from right to left + 1 step −1
      if A[i] < A[i − 1]
        swap A[i] and A[i − 1]
    left = left + 1
```

2.2 Searching Algorithms

2.2.1 Linear Search

In this searching algorithm we iterate through the dataset one element at a time, comparing each element with the target item until you find a match or reach the end of the dataset.

Pseudo Code

```
Procedure LinearSearch(A: array of items, target: item): integer
  for i = 0 to length of A − 1
    if A[i] = target
      return i // Target found at index i
  return −1 // Target not found
```

2.3 Multi Level Sorting

Multi-level sorting is a technique employed to arrange data in a structured order based on multiple criteria. Picture a list of items with various attributes like names, dates, and reviews. With multi-level sorting, you can sort these items first by one attribute (e.g., alphabetically by name) and then, if there are duplicate values in the first attribute, sort them further by another attribute (e.g., numerically by reviews). This approach allows you to organize your data hierarchically, ensuring that it's presented in a specific sequence that meets your requirements.

Pseudo Code

```
Procedure MultiColumnMergeSort(arr: array of items, start: integer, end: integer, columns: list of columns)
  if start < end
    mid ← start + ⌊(end−start)/2⌋
    MultiColumnMergeSort(arr, start, mid, columns)
    MultiColumnMergeSort(arr, mid + 1, end, columns)
    MultiColumnMerge(arr, start, end, mid, columns)

Procedure MultiColumnMerge(arr: array of items, start: integer, end: integer, mid: integer, columns: list of columns)
  left ← arr[start : mid + 1]
  right ← arr[mid + 1 : end + 1]

  i ← 0
  j ← 0
  k ← start

  while i < length(left) and j < length(right)
    if MultiLevelCompare(left[i], right[j], columns)
      arr[k] ← left[i]
      i ← i + 1
    else
      arr[k] ← right[j]
      j ← j + 1
    k ← k + 1

  while i < length(left)
    arr[k] ← left[i]
    i ← i + 1
    k ← k + 1

  while j < length(right)
    arr[k] ← right[j]
    j ← j + 1
    k ← k + 1

Procedure MultiLevelCompare(leftArray: array of items, rightArray: array of items, columns: list of columns)
  for each col in columns
    left_val ← leftArray[col]
    right_val ← rightArray[col]

    if is_numeric(left_val) and is_numeric(right_val)
      if left_val < right_val
        return True
      else if left_val > right_val
        return False

    else if is_string(left_val) and is_string(right_val)
      if left_val < right_val
        return True
      else if left_val > right_val
        return False

  return False
```

2.4 Multi Column Searching

Multi-column searching is a method used to find specific information in a database or data set. Imagine you have a large table with different types of data like names, dates, and numbers. Instead of searching in just one column (like looking only at names), multi-column searching allows you to look for something using multiple columns at once. For example, you could search for people with a specific name who also bought a certain product within a particular time frame. This kind of searching is very flexible and helps you find exactly the information you need from a big pool of data.

2.5 Advanced Search and Filters

User can also do an advanced search with following parameters:

1. And
2. Not
3. Or

3 User Interface and Functionalities

3.1 Main Window

The Main Window has four sections:

1. **Left Panel:** The left panel has three button. Scrap Data button is used for scrapping data for our app from APK Pure's Website. Multi-Columns-Searching button opens the "Multi Column Search Window. The Exit Button is used for closing the application. It also has Run Time box which is used for showing run time of algorithms."
2. **Top Panel:** The top panel is divided into three parts. The left part has three buttons with which user can sort data by choosing the desired column, order and algorithm. The middle part has one button which is used for refreshing the table output. The right part has two button which open two new windows for performing the operations specified on them.
3. **Bottom Panel:** The bottom panel has progress bar and three buttons for controlling the scrapping operations.
4. **Main Panel:** The main panel has table view widget which is used for displaying the data from CVS file.

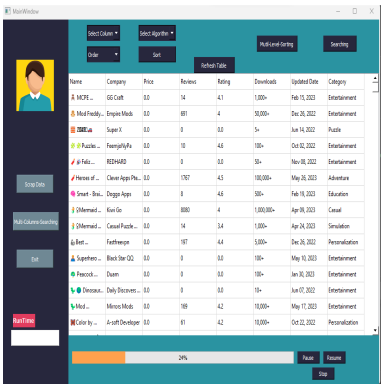

Figure 1: Main Window

3.2 Multi Column Sort Window

This window has eight check boxes and two buttons. The check boxes are used for selecting columns for multi column search. The exit button is used for closing the window while Sort button is used for performing sorting opertaion.

Figure 2: Multi Column Sort Window

3.3 Multi Column Search Window

This window has same features as Multi Column Sort Window additionally it has Select Filter drop down for applying filter and one text box for searching the desired value.

Figure 3: Multi Column Search Window

3.4 Search Output Window

The Search Output Window has two drop downs, one for selecting column for searching and the other applying appropriate filer. The text box is used for searing specific value while exit button closes the window. The search button starts the search operation.

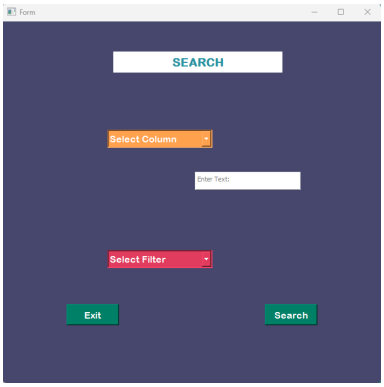

Figure 4: Search Output Window

4 Conclusion

We successfully achieved our data scraping and sorting goals with a user-friendly GUI. We collected data from APK Pure, developed an efficient user interface, and identified Merge Sort as the most effective sorting algorithm. For integers, non-comparison-based algorithms like Counting Sort, Pigeon Hole Sort, and Radix Sort were optimal. Future plans include enhancing user interface re-

sponsiveness and giving user more control over data. In conclusion, this project helped to learn practical implementation of algorithms and it a great tool for data analysis.