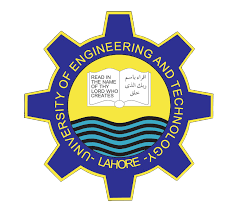
****University of Engineering and Technology

**Song Record**

PlayListical

The Desktop Application Based On Web Scraping

**Group Members**

2020-CS-70

2020-CS-97

**Group ID**

CS261F21PID25

Department of Computer Science

**Supervisor:** Samyan Qayyum Wahla

Table of Contents

**Problem Statement1**

**Project Description1**

**Learning Outcomes2**

**Efficiency2**

**Business Case2**

Need for Project 2

End user of Product2

Motivation 2

Level of Impact2

**Project Features3**

Sorting Order3

Searching Algorithm3

**Details of Features4**

Sorting Order4

Searching Algorithm4

**Requirements5**

Sorting Algorithm5

**Details of Algorithms6**

**Insertion Sort8**

**Selection Sort10**

**Merge Sort13**

**Bubble Sort14**

**Quick Sort15**

**Heap Sort10**

**Count Sort13**

**Radix Sort14**

**Shell Sort15**

**Attributes of Entity20**

**Technical Details21**

**Technology Stack21**

**Use Case28**

**UI Component Details30**

**Project Plan36**

**Problem Statement**

Music is very pleasing to human ears. We listen to music when we are stressed, happy and sad. Now days, Music industry is growing day by day. And to provide the track of each and every album, we are designing a desktop based application using web scrapping. This will allow the users to find any kind of song relating to year, album or artist to find easily. Millions of Data will be scrapped and we will allow user to load data into application according to their need. They can easily start, stop and pause scrapping process, depending on their need.

**Project Description**

The proposed Song Record (Playlist) is a Desktop-based Python project. It is a comprehensive playlist system that will include inventory of million songs by Web-Scrapping. This will include sorting songs alphabetically in ascending or descending order. Similarly, searching any particular song by name and specific column.

About a million entities will be scrapped in this project form different websites to make a playlist of songs which a user can sort using multiple sorting algorithms i.e. User will have the choice to choose any sorting algorithm to sort a particular column along with an option to sort ascending or descending. User will also have an option to sort using multiple columns. Time taken for each sorting algorithm will be displayed in milliseconds after sorting has been done. List of sorting algorithms will be mentioned later.

User will also be given access to pause, start, resume and stop the scrapping tasks with the progress bar showing the progress of tasks/ number of entities scrapped. So that the user can pause or stop at any specific time.

User will be able to sort data using filters (available options) to each of the entities column. Project will contain advanced filters for columns such as contains, end with, starts with etc. Project will also have a feature to search, based on each column. User will also have the option to search using composite filters such as AND, OR and NOT.

**Learning Outcome**

By doing this project we will be able to:

* Scrap millions of data from website.
* Manage the scrapped data in a ‘CSV’ file.
* Read data from ‘CSV’ file
* Apply algorithms on exceedingly large data.
* Apply filters such as Ascending/ Descending.

**Efficiency**

The algorithms we are using in this project are adjustably efficient. Their time complexity is explained below.

**Business Case**

**Need for Project**

Artists use a technique called **social media tracking.** This allows them to track mentions, comments, likes. It helps the Music companies to analyze trends and predict what the next big hit may be.

**End user of Product**

This application can be extremely useful for Music Data Analyzers. It can provide them with insights, pros and cons; also it can help the analyzers to depict the most liked songs.

**Motivation**

Firstly, this project will save your time in searching. Secondly, it will contain every type of sorting methods. This will allow the users to find any kind of song relating to year, album or artist to find easily. Millions of Data will be scrapped and we will allow user to load data into application according to their need.

**Level of impact**

Project will be really impactful for the music industry and song analyzers and if we are not proceeding then, we will look upon our project again.

**Project Features**

**Sorting Order:**

1. Ascending Order
2. Descending Order

**Search Algorithm:**

1. Binary Order
2. Linear Order

**Filters:**

1. Greater than
2. Less than
3. Equal to
4. Contains with
5. Starts with
6. Ends with

**Details of Features**

**Sorting Order**

**Ascending Order:** This filter will help us to implement each of our algorithms in Ascending Order.

**Descending Order:** This filter will help us to implement each of our algorithms in descending Order.

**Search Algorithm**

Binary Search and Linear Search both will be used in this application; however binary search will be use more frequently because of its efficient result. User can search any of the entities from the attribute and can also add filters.

**Greater than:** This filter will allow us to find the greater values of the entered input by implementing our search algorithms.

**Less than:** This filter will allow us to find the lesser values of the entered input by implementing our search algorithms.

**Equal to:** This filter will allow us to find the values equal to that of entered input by implementing our search algorithms.

**Starts with:** This filter will allow the user to enter an alphabet to find song, album, singer, etc. that begins with that specific letter.

**Ends with:** This filter will allow the user to enter an alphabet to find song, album, singer, etc. that ends with that specific letter.

**Contains:** This filter will allow the user to enter an alphabet to find song, album, singer, etc. that contains the specific letter.

User will also have the option to search multiple level searching using composite filters such as AND, OR and NOT

**Requirements**

**Sorting Algorithms**

Listed below are the algorithms which we will use on our filters to give desired result to the user:

1. Insertion Sort
2. Merge Sort
3. Selection Sort
4. Bubble Sort
5. Quick Sort
6. Radix Sort
7. Counting Sort
8. Heap Sort
9. Shell Sort

User will have the choice to sort the data using any algorithms given above for sorting of a particular column. All these sorting Algorithms will sort in ascending and descending order.

**Details of Algorithm**

|  |  |
| --- | --- |
| **Algorithm Name** | **Description(Each algorithm in 2-3 lines)** |
| Insertion Sort | Insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. |
| Merge Sort | Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several arrays until each arrays consists of a single element and merging those arrays in a manner that results into a sorted list. |
| Selection Sort | In Selection sort we divide the array into two parts: sorted and unsorted. The left part is sorted subarray and the right part is unsorted subarray. Initially, sorted subarray is empty and unsorted array is the complete given array. |
| Bubble Sort | It is simple sorting algorithm that repeatedly goes through the list, compares adjacent elements and swaps them if they are in the wrong order. This is the most simplest algorithm and inefficient at the same time. |
| Quick Sort | Quick sort is capable of sorting a list of data elements significantly faster (twice or thrice faster) than any of the common sorting algorithms. and is based on the splitting of an array (partition) into smaller ones and swapping (exchange) based on the comparison with 'pivot' element selected |
| Radix Sort | Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort. |
| Counting Sort | Counting Sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence. |
| Heap Sort | Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements |
| Shell Sort | Shell sort improves on the insertion sort by breaking the original list into a number of smaller array, each of which is sorted using an insertion sort. The unique way that these array are chosen is the key to the shell sort. |

**Insertion Sort**

**Description**

Insertion sort is a mechanism of sorting in which sorted array is constructed one element at a time. The elements of an array are compared to each other sequentially and then arranged. Start with an ordered array of 1 item on the left and N-1 unsorted items on the right. Take the first unsorted array (item 2) and insert it into the ordered array, shifting the items as needed. Now we have an ordered list of unsorted items in size 2 and N -2. Repeat for all items. It’s a bit like bubble sort for moving items, except when it finds an item smaller than you, it stops. If the data is sorted backwards, each item must go to the top of the list, which becomes a bubble sort. There are several ways to move an element to the left: you can swap each iteration or copy each element to your neighbor.

|  |  |
| --- | --- |
| Pseudo code | def Insertion\_Sort(array):  int start  int insert    for i in range(1,len(A)):  insert = A[i]  start = i    while start > 0 and A[start-1] > insert:  A[start] = A[start -1]  start = start -1    A[start] = insert |
| Code In Python | def insertionSortFruits():  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=j-1  array[j+1]=key  print(array) |
| Time complexity | **Best:** O(n)  **Worst:** O(n^2) |
| Dry Run | input:  0 1 2 3 4  2 1 0 11 9  When i=0  0 1 2 3 4  2 1 0 11 9  When i=1  0 1 2 3 4  1 2 0 11 9  When i=2  0 1 2 3 4  0 1 2 11 9  When i=3  0 1 2 3 4  0 1 2 11 9  When i=4  0 1 2 3 4  0 1 2 9 11 |

**Proof of correctness**

Many proof techniques, such as contradiction proof and induction proof, can be used to prove that the insertion sort is correct, we will use "loop invariants". We need to test three conditions:

1. Initialization

2. Maintenance

3. Termination

**Initialization:** before the first iteration, when j = 2, the sub array [1 to j - 1] is only the first element of the array, A [1]. This sub array is ordered and consists of the elements that were originally in A [1to 1].

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

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

**Strengths**

1. The main advantage of insertion sort is its simplicity and clearness.
2. It also works well for a minimum and small array.
3. Insertion sort is an in-place sort algorithm, so the space requirement is negligible

**Weaknesses**

1. The downside to insertion sort is that it doesn't work as well as the better sort algorithms.
2. Since steps of n squares are required for each n element to be sorted, insertion sort does not work well with a large list.
3. Insertion sort is especially useful only when ordering a list of multiple items.

**Selection sort**

**Description**

In Selection Sort, we divide the array into two parts: ordered (sorted) and unordered (unsorted). The left part is an ordered sub array and the right part is an unsorted sub array. The initial ordered sub array is empty and the unsorted array is the given full array. Scan all items and find the smallest. Change it to its position as the first item. Repeat the selection sort for the remaining N-1 items. This is the easy to implement as it always iterates forward and swaps with the smallest element.

|  |  |
| --- | --- |
| Pseudo Code | List = len(A)  n = len(list)  for i = 1 to n - 1  mini = i  for j = i+1 to n  if list[j] < list[mini]:  mini = j;  if index\_mini != i:  swap list[mini] and list[i] |
| Code in Python | for i in range(len(A)):  min\_idx = i  for j in range(i+1, len(A)):  if A[min\_idx] > A[j]:  min\_idx = j  A[i], A[min\_idx] = A[min\_idx], A[i] |
| Time Complexity | **Best:** O(n^2)  **Worst:** O(n^2) |
| Dry Run | input:  0 1 2 3 4  2 45 0 11 9  When i=0  0 1 2 3 4  2 45 0 11 9  When i=1  0 1 2 3 4  0 45 2 11 9  When i=2  0 1 2 3 4  0 2 45 11 9  When i=3  0 1 2 3 4  0 2 9 11 45  When i=4 no iteration is required as the last element is already sorted  0 1 2 3 4  0 2 9 11 45 |

**Proof of correctness**

Many proof techniques, such as contradiction proof and induction proof, can be used to prove that the selection sort is correct, we will use "loop invariants". The loop invariant we will use is

we need to test three conditions:

1. Initialization

2. Maintenance

3. Termination

This is essentially a mathematical induction, initialization is the base case and maintenance is the inductive step,

**Initialization:** Before the first iteration of the loop, j = i + 1. Then, the array segment A [i..j-1] it's just point A [i]., we have min indexes the smallest element (the only element) in the sub array A [i..j-1] and invariant is true.

**Maintenance:** Before passing j, we assume that min indexes the smallest element of the sub array. A [i..j-1]. During iteration j we have two cases:

**A [j] <A [min]**

or A [j] A [min]. In the second case, the if statement is not true, therefore nothing is executed. But now min indexes the smallest element of A [i..j]. In the first case, changes min at the location of index j, because it is the smallest. If min indexes an element less than or equal to the sub array A [i..j-1] and now

**A [j] <A [min]**

then A [j] must be less than or equal to elements of sub-array A [i..j-1]. changes min to index this new location and so after the end of the iteration of the loop, min indexes the smallest element in sub-table A [i..j].

**Termination:** At the end of the inner loop, min indexes an element less than or equal to all elements of the sub-array A [i..n] Since j = n + 1 after completion. This finds the

smallest element of this sub array and is useful to us in the outer loop because you can move the next smaller item to the right location.

**Strength**

1. The main benefit of sorting by selection is that it works well for a small list.
2. This is an in-place sort algorithm; no additional temporary storage is required beyond what is needed to store the original list.
3. Its performance is easily influenced by the initial ordering of the items before the sorting process.

**Weakness**

1. The main disadvantage of the selection sort is its low efficiency when it comes to a large list of items.
2. Your performance is easily affected by the initial order of items prior to the sorting process. For this reason, sorting by selection is only appropriate for a list of certain items that are in random order.
3. The selection sort requires n-squared number of steps for sorting n elements.

**Merge Sort**

**Description**

Merge sort is a divide and conquer algorithm. Divides the input array into two arrays and calls recursively itself for the two arrays, and then joins the two ordered arrays. Most of the algorithm gets two ordered arrays, and we need to merge them into one ordered array. The entire process of ordering an array of N integers can be summarized in three steps:

1. Divide the matrix into two portions.
2. Sort the left portion and right portion using the same repeating algorithm.

Combine the ordered portions together.

|  |  |
| --- | --- |
| Pseudo Code | mergesort( Array ):  if ( n == 1 )  return Array  arr1 as array = a[0] ... a[n/2]  arr2 as array = a[n/2+1] ... a[n]  arr1 = mergesort( arr1 )  arr2 = mergesort( arr2 )  return merge(arr1, arr2 )  merge( a, b )  while ( a and b consists ofelements )  if ( a[0] > b[0] )  add b[0] to the end of c  remove b[0] from b array  else  add a[0] to the end of c array  remove a[0] from a array    while ( a has elements )  add a[0] to the end of c  remove a[0] from a array    while ( b has elements )  add b[0] to the end of c  remove b[0] from b array    return c |
| Code in Python | def mergeSort(List):  if len(myList) > 1:  mid = len(List) // 2  left = List[:mid]  right = List[mid:]  mergeSort(left)  mergeSort(right)  i = 0  j = 0  k = 0    while i < len(left) and j < len(right):  if left[i] <= right[j]:  List[k] = left[i]    i += 1  else:  List[k] = right[j]  j += 1  k += 1  while i < len(left):  List[k] = left[i]  i += 1  k += 1  while j < len(right):  List[k]=right[j]  j += 1  k += 1 |
| Dry run | input  0 1 2  5 9 11  Splitting [5, 9, 11]  Splitting [5]  Merging [5]  Splitting [9, 11]  Splitting [9]  Merging [9]  Splitting [11]  Merging [11]  Merging [9, 11]  Merging [5, 9, 11] |
| Time Complexity | **Best:** O(n log n)  **Worst:** O(n log n |

**Proof of correctness**

For the base case, consider a 1-element array. Such a matrix is ​​already ordered, so the base case is correct. For the induction step, assume that Merge Sort correctly sorts any array of length less than n. suppose we call Merge Sort on an array of size n. Will recursively call Merge Sort in half of the arrays? Under the induction hypothesis, these calls will correctly order these matrices. For that, after the recursive calls, the array A will be sorted between the indices a . . . l and l + 1 . . . z respectively.

We have already shown that the merge sort works fine, so after performing it, A array will be fine ordered between a and z. This concludes our proof.

**Strength**

1. It can be used for data of any size.
2. It is faster for larger array because, unlike inserting and sorting bubbles, it doesn't make the entire program go through the whole array multiple times.
3. It has a fixed execution time, executes different bits with similar times in one step.

**Weaknesses**

1. Slower comparison with other sort algorithms for smaller tasks.
2. It goes through the whole process even if the list is sorted
3. It uses more memory space to store the sub elements of the initial split list.

**Bubble Sort**

**Description**

This sorting algorithm moves through the array repeatedly, comparing pairs of elements and swapping their positions if they are in the wrong order. The algorithm keeps going through the list in this way until the whole list is sorted.

Starting from the left, compare adjacent items and continue "Swapping" the largest one on the right in its final position. Sort the rest of the N-1 items with bubbles. With the bubble sort, you can bubble items "forward" (left to right) and move the endpoint backward (decreasing) or bubble items "backward" (right to left) and increase the left endpoint. Either way, some indexes are going down. You should also keep track of the next-to-last endpoint to avoid becoming an item that doesn't exist.

|  |  |
| --- | --- |
| Pseudo Code | Bubble\_Sort(list):  n = list.count;  for i in range(0,n-1):  S = false  for j in range(0,n-1):  if list[j] > list[j+1] then  swap( list[j], list[j+1] )  S= true  if(!S):  break  return list |
| Code in Python | def bubble\_sort(arr):      n = len(arr)      for i in range(n):          for j in range(n - i - 1):              if arr[j] > arr[j + 1]:                  # sorting by using simultaneous assignment in python                  arr[j], arr[j + 1] = arr[j + 1], arr[j]      return arr |
| Dry run | input:  0 1 2 3 4  10 43 4 9 5  When i=0  0 1 2 3 4  10 43 4 9 5  When i=1  0 1 2 3 4  10 4 9 5 43  When i=2  0 1 2 3 4  4 9 5 10 43  When i=3  0 1 2 3 4  4 5 9 10 43  When i=4  0 1 2 3 4  4 5 9 10 43 |
| Time Complexity | **Best:** O(n)  **Worst:** O(n^2) |

**Proof of correctness**

Validating bubble sort is the same as sorting a selection. It first finds the smallest element and switches it to the array entry of the 0th. Then it finds the 2nd smallest element and swaps it to array of the entry 1, then the 3rd element, the 4th , and so on. It never invalidates previous work. After nth iterations through this loop on an array containing n elements, we have a completely ordered array

**Strengths**

1. The main advantage of bubble sort is popularity and ease of implementation.
2. The items of array are swapped into place without the use of additional temporary storage, so the usage of memory space requirement is minimal.
3. The space requirement is at a minimum

**Weaknesses**

1. Disadvantage of bubble sort is that it doesn't perform well with an array that has a large number of elements.
2. This is because bubble sort requires n square processing steps for every n items to be sorted.
3. The bubble type is best suited for academic teaching, but not for real-world applications.

**Quick Sort**

**Description**

Quick sort is able to sort a list of data items much faster (two to three times faster) than any of the popular sort algorithms. and consists in dividing the array (partition) into smaller ones and replacing (swapping) on ​​the basis of comparison with the selected 'pivot' element

Pick a pivot and swap it

Going left to right, find the element that is larger than the pivot

Going right to left, find the element that is smaller than the pivot

Change items if you find them and continue until the pivot

If you choose the wrong pivot, that there is "small" subset is always empty. Means that we are creating a sub array of the smaller element at a time, which gives us O (N ^ 2) behavior in the worst case. You can choose a random item or the median of three (front, middle, end).Quicksort is fast because it goes through adjacent elements, comparing them to a pivot value

**Proof of correctness**

Many proof techniques, such as contradiction proof and induction proof, can be used to prove that the heap sort is correct, we will use "loop invariants". The loop invariant we will use is

we need to test three conditions:

1. Initialization

2. Maintenance

3. Termination

This is essentially a mathematical induction, initialization is the base case and maintenance is the inductive step,

Initialization: before the first iteration. If everything is a leaf so it is already a heap.

Maintenance: Suppose solution that works so far have the children of node x are numbered higher than x. Heap Sort also preserves the invariant loop. We maintain invariance at every step.

Termination: terminates when x drops to 0 and through the invariant loop, each node is the root of a maximum heap.

**Strengths**

1. The algorithm is efficient. The performance is optimal. This means that no other sorting algorithm can perform better in comparison.
2. Memory consumption is minimal because apart from what is necessary to maintain the initial list of items to sort, it does not require additional memory to function.
3. The heap sort algorithm shows consistent performance. This means it works equally well for best, average, and worst case scenarios.

**Weaknesses**

1. A stable sort maintains the relative order of items that share the same key. Heap sort is an unstable sort. It can change the relative order.
2. In real implementations, there are constant factors that theoretical analysis does not take into account. In the case of Heap sort, it turns out that there are ways to make the worst cases of Quicksort very rare.
3. Quicksort will run faster than Heap sort because its fixed coefficients are smaller Heap sort. In Simple words, partitioning is faster than keeping the heap.

**Heap Sort**

**Description**

Heap Sort is a comparison-based sort technique that is based on the binary structure of the heap data. This is similar to sorting by selection where we find the minimum element first and put the minimum element first. We repeat the same process for the rest of the items Add all items to the heap. Take the minimum item out of the stack and add it first Repeat for all items. Heapsort is like selection sort, but with a better way to get the minimum item. Instead of scanning each item to find the minimum, it pulls it out of the heap. Heaps have properties that allow you to work in place without additional memory. The heap formation is O (N lg N). The element popping is O (1) and the arranging of the heap after breaking is lgN. There are N pops, so there is another O (N lgN) factor which is generally O (N lg N).Heapsort has O (N lgN) behavior, even in the worst case, which makes it good for real-time applications

|  |  |
| --- | --- |
| Pseudo Code | for i = 1 to size:        node = i        p = floor (node / 2)        while p >= 1:           if array[par] < array[node]              swap array[p] with array[node]           node = p           p = floor (node / 2) |
| Code in Python | def heapify(arr, n, i):  largest = i # Initialize largest as root  l = 2 \* i + 1 # left = 2\*i + 1  r = 2 \* i + 2 # right = 2\*i + 2  if l < n and arr[i] < arr[l]:  largest = l  if r < n and arr[largest] < arr[r]:  largest = r  if largest != i:  arr[i],arr[largest] = arr[largest],arr[i] # swap  heapify(arr, n, largest)  def heapSort(arr):  n = len(arr)  for i in range(n // 2 - 1, -1, -1):  heapify(arr, n, i)  for i in range(n-1, 0, -1):  arr[i], arr[0] = arr[0], arr[i] # swap  heapify(arr, i, 0) |
| Dry Run | input:  0 1 2 3  11 34 9 5  When i=4  0 1 2 3  34 11 9 5  When i=3  0 1 2 3  5 11 9 34  When i=2  0 1 2 3  11 5 11 34  When i=1  0 1 2 3  9 5 11 34  When i=0  0 1 2 3  5 9 11 34 |
| Time Complexity | **Best:** O(n log n)  **Worst:** O(n log n) |

**Proof of Correctness**

Many proof techniques, such as contradiction proof and induction proof, can be used to prove that the heap sort is correct, we will use "loop invariants". The loop invariant we will use is

we need to test three conditions:

1. Initialization

2. Maintenance

3. Termination

This is essentially a mathematical induction, initialization is the base case and maintenance is the inductive step,

**Initialization**: before the first iteration. If everything is a leaf so it is already a heap.

**Maintenance:** Suppose solution that works so far have the children of node x are numbered higher than x. HeapSort also preserves the invariant loop. We maintain invariance at every step.

**Termination:** terminates when x drops to 0 and through the invariant loop, each node is the root of a maximum heap.

**Strengths**

1. The algorithm is efficient. The performance is optimal. This means that no other sorting algorithm can perform better in comparison.
2. Memory consumption is minimal because apart from what is necessary to maintain the initial list of items to sort, it does not require additional memory to function.
3. The heap sort algorithm shows consistent performance. This means it works equally well for best, average, and worst case scenarios.

**Weakness**

1. A stable sort maintains the relative order of items that share the same key. Heapsort is an unstable sort. It can change the relative order.
2. In real implementations, there are constant factors that theoretical analysis does not take into account. In the case of Heapsort, it turns out that there are ways to make the worst cases of Quicksort very rare.
3. Quicksort will run faster than Heap sort because its fixed coefficients are smaller Heapsort. In Simple words, partitioning is faster than keeping the heap.

**Counting Sort**

**Description**

Counting sort is a sort technique based on keys in a specific range. It works by counting the number of objects that have different key values . Then do arithmetic to calculate the position of each object in the output sequence.

Assuming the data are integers ranging from 0-k. Create a size K array to keep track of the number of items displayed. Given this number, you can know the position of the element: all 1s must come after the zeros So we can scan the items and put them in the right place. The creation of the count array is O (N). Inserting items in the correct position is O (N)

I simplified here: there is the sum of the counts and the order from highest to lowest, which keeps the sort stable.

|  |  |
| --- | --- |
| Pseudo Code | max = get maximum element from array.  Count = [0]\*max+1     for i in range(0,max):  count[i] = 0     for i in range(0,max):  increase count     for i in range(0,max):        count[i] = count[i] + count[i+1]     for i in range(size, size-1):        store the number in the output array        decrease count[i]     return array |
| Code in Python | def countSort(arr):      output = [0 for i in range(len(arr))]      count = [0 for i in range(len(output))]      for i in arr:          count[ord(i)] += 1      for i in range(256):          count[i] += count[i-1]      for i in range(len(arr)):          output[count[ord(arr[i])]-1] = arr[i]          count[ord(arr[i])] -= 1      for i in range(len(arr)):          ans[i] = output[i]      return ans |
| Time Complexity | **Best:** O (n+k)  **Worst:** O(n+k) |
| Dry Run | **Counting Sort Algorithm Dry Run**  input:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 2 | 1 | 1 | 4 | 2 |   Count Array   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 0 | 0 | 0 | 0 |   frequency of every element   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 2 | 2 | 0 | 1 |   cumulative sum   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 2 | 4 | 4 | 5 |   Now start iterating input array and check frequency array to map it to the output array  When i=1  input:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Index | 0 | 1 | 2 | 3 | 4 | | Count | 0 | 2 | 3 | 4 | 5 | | Output | 0 | 0 | 0 | 2 | 0 |   Iteration 2  input:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Index | 0 | 1 | 2 | 3 | 4 | | Count | 0 | 1 | 3 | 4 | 5 | | Output | 0 | 1 | 0 | 2 | 0 |   Iteration 3  input:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Index | 0 | 1 | 2 | 3 | 4 | | Count | 0 | 0 | 3 | 4 | 5 | | Output | 1 | 1 | 0 | 2 | 0 |   Iteration 4  input:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Index | 0 | 1 | 2 | 3 | 4 | | Count | 0 | 0 | 3 | 3 | 4 | | Output | 1 | 1 | 0 | 2 | 4 |   Iteration 5  input:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Index | 0 | 1 | 2 | 3 | 4 | | Count | 0 | 0 | 2 | 3 | 4 | | Output | 1 | 1 | 2 | 2 | 4 | |

**Proof of Correctness**

**Initialization**: Suppose that the positions i and j with i <j contain an element k.

**Maintenance:** We consider COUNTINGSORT, where we construct the output array. Since j> i, the loop will examine A [j] before examining

**Termination**: When this is the case, the algorithm correctly places A [j] at the position m = C [k] of B. Since C [k] decreases and never increases again, we are assured that when the for loop examines At [i] we are going to have C [k] <m. Therefore, A [i] will be placed earlier in the output array, demonstrating stability.

**Strengths**

1. Linear time complexity. Since this is not a comparison based classification, it is not limited by the O (nlogn) complexity.
2. Reducing the complexity of the space if the range of elements is narrow, i.e. higher frequency of close integers.
3. It is stable algorithm.

**Weakness**

1. Both temporal and spatial complexity increase sharply if the input number range is large.
2. Only works for discrete values ​​such as integers.
3. With negative integers, complexity increases and some algorithm changes are required

**Radix Sort**

**Description**

Radix sort is a digit by digit sort starting from the least significant digit to the most significant digit. Radix Sort uses count sort as the sort routine.

Get a series of numbers and arrange them one digit at a time. Repeat the sort for each set of digits.

Radix sort uses count sort to efficiently sort O (N) digits (k = 0 ... 9)

Sorting by count and Radix is fast, but requires structured data, external storage, and doesn't have the advantages of fast cache sort

|  |  |
| --- | --- |
| Pseudo Code | 10 list     for i := 0 to max -1 do  m = 10^i+1        p := 10^i        for j := 0 to n-1 do           temp := array[j] mod m           index := temp / 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(array, place):  size = len(array)  output = [0] \* size  count = [0] \* 10  for i in range(0, size):  index = array[i] // place  count[index % 10] += 1  for i in range(1, 10):  count[i] += count[i - 1]  i = size - 1  while i >= 0:  index = array[i] // place  output[count[index % 10] - 1] = array[i]  count[index % 10] -= 1  i -= 1  for i in range(0, size):  array[i] = output[i] |
| Time Complexity | **Best:** O(nk)  **Worst:** O(nk) |
| Dry Run | input:  0 1 2 3 4  244 153 271 141 900  Sorting by least significant digit  0 1 2 3 4  900 271 141 153 244  Sorting by 2nd least significant digit  0 1 2 3 4  900 141 244 153 271  Sorting by Most significant digit  0 1 2 3 4  141 153 244 271 900 |

.

**Proof of Correctness**

**Loop Invariable**: At the start of the for loop, the array is sorted in the last n - 1 digits.

**Initialization:** The array is sorted trivially in the last 0 digits.

**Maintenance:** Suppose the array is ​​ordered by the last n- 1 digits. After sorting by the nth digit, the array will be sorted by the last n digits. When the elements with different the digits in nth position are sorted accordingly; in the case of the same nth digit we still get correct ordering of digits , because we are using stable sorting and the items were already ordered in the last n - 1 digits.

**Termination**: The loop ends when n = d + 1. Since the invariant holds, we have the numbers ordered in d digits

**Strengths**

* 1. Fast when the keys are short, that is, when the range of array elements is smaller.
  2. It is used in suffix array building algorithms such as the Manber algorithm and the DC3 algorithm.
  3. Radix Sort is a stable sort because the relative order of items with equal values ​​is maintained.

**Weakness**

1. Since Radix Sort is based on numbers or letters, Radix Sort is much less flexible than other types of sorting algorithms. Therefore, for any other type of data, it is necessary to rewrite them.
2. Radix sort is slower than sorting algorithms, such as merge sort and quick sort
3. This is not an in-place sorting algorithm as it requires additional space for memory usage

**Shell Sort**

**Description**

Shell sort is an efficient sorting algorithm those properties based on an insertion sort algorithm. This algorithm avoids big intervals

This algorithm uses insertion sort on interval, first to sort them, and then to sort the items with the less spacing. This algorithm is a bit of efficient for average amount of dataset because its average complexity and in the worst case this algorithm depends on the order of the spaces, the best known is Ο (n), where n is the number of items in an array. And complexity of the worst case is O (n).

In simple word, Split the array into a smaller array with an interval equal to h, sort these subarrays with insertion sort, repeat until you sort the entire array

|  |  |
| --- | --- |
| Pseudo Code | shellSort()  A =[]  while mid < len(A) /3 do:  mid = mid \* 3 + 1  while mid > 0 do:  for outer = mid;  outer < len(A); outer ++ do:  Insert = A[outer]  inner = outer;  while inner > mid -1 && A[inner - mid] >= Insert do:  A[inner] = A[inner - mid]  inner = inner – mid  A[inner] = Insert  mid = (mid -1) /3; |
| Code in Python | def shellSort(array, n):      # Rearrange elements at each n/2, n/4, n/8, ... intervals      interval = n // 2      while interval > 0:          for i in range(interval, n):              temp = array[i]              j = i              while j >= interval and array[j - interval] > temp:                  array[j] = array[j - interval]                  j -= interval              array[j] = temp          interval //= 2 |
| Time Complexity | **Best:** O(n log n)  **Worst:** O(n log n ^2) |

**Proof of Correctness**

To show that shell sorting actually sort the array in sorted order,

The shell sort has an outer loop that determines a sequence of spaces this sequence ends with 1.

When the space is 1, the two inner loops become equivalent to an insert sort. So,

the fact that shell sorting is a valid sorting algorithm is based on these facts:

**Initialization:** where the difference is greater than one, the algorithm only permutes the input array; it does not add, remove, or modify keys.

**Maintenance:** The result of ordering an array is ​​the same as that of ordering a permutation of the matrix.

**Termination:** the space is equal to one, the algorithm is reduced to sort by insertion.

**Strengths**

1. Due to the increased complexity of average case, it is a very efficient algorithm for medium-sized arrays.
2. Better than insertion sort and 5 times faster than bubble sort.
3. Shell sort is a highly efficient sorting algorithm based on an insertion sort algorithm

**Weakness**

1. It is a complex algorithm.
2. Not as efficient as a Merge sort and quick sort.
3. Limited use on small arrays as performance degrades as the array size increases.

**Attributes of Entity**

Application will contain several columns, based on these columns the user can perform multiple operations i.e. sorting with multiple algorithms, option to sort using multiple columns, search based on each column, filters to each of the entities column

Application will contain the columns given below:

1. Song Title
2. Artist
3. Album
4. Genre
5. Year
6. Duration
7. Language
8. Likes

As each column name suggested it will contain detail about each entity.

**Details**

**Song Name** attribute will contain the name of the song and it can be sorted and searched alphabetically and multiple filters can also be added.

**Artist** attribute will contain the name of the Singer/Band and it can be sorted and searched alphabetically and multiple filters can also be added.

**Album** attribute will have the album name that song is form and it can also be sorted and searched alphabetically and multiple filters can also be added.

**Genre** will contain the genre of the song.

**Year** will contain era of the song i.e. 80’s or 90’s. It can be sorted era wise or filters can be added like era of 80’s or era greater than 80’s.

**Duration** will have the time duration of the song; this attribute will contain filters to search song greater than 2 min or less than 3 min

**Language** will contain the specific name of the language in which song is sung. It can be sort and search alphabetically.

**Likes** which contains total number of likes that song have and this attribute can be sorted number wise or filters such as likes greater than 20,000.

**Technical Details**

|  |  |  |
| --- | --- | --- |
| *Name* | *Data Type* | *Description* |
| Title | String | Name of the song |
| Album | String | Name of the Album |
| Artist | String | Name of the Artist |
| Posted | String | When that song was posted |
| Likes | integer | Number of likes that song have |
| Comment | integer | Number of comments |
| Of times Played | String | How many time that song have been played |
| Genre | String | Genre of that song |

**Technology Stack**

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

**Use Cases**

This part of report describe the concept of project using user interface

## Use Case 1:

|  |  |
| --- | --- |
| Use Case ID | U1 |
| Name | Main Page |
| Actor | User |
| Description | This screen will allow the user to scrap data from the file containing millions of data relating to songs. And the user will have access to start, stop and resume the scrapping process from file to the table until the required result has been achieved. Then next screen will pop |
| GUI |  |
| Validators | None |

## Use Case 2:

|  |  |
| --- | --- |
| Use Case ID | U2 |
| Name | Next Screen |
| Actor | User |
| Description | This screen will allow the user to arrange the data using his\her own requirements. This screen will contain filter on the header of each column and there will be more than 5 columns user will be able to trigger the arrangement of the data accordingly. |
| GUI |  |
| Validators | None |

## Use Case 3:

|  |  |
| --- | --- |
| Use Case ID | U3 |
| Name | Drop Down |
| Actor | User |
| Description | This is used to contain all the features applicable on this project. This drop down will allow us to ascend, descend and filter the data present in table. |
| GUI |  |
| Validators | None |

## Use Case 4:

|  |  |
| --- | --- |
| Use Case ID | U4 |
| Name | Filters |
| Actor | User |
| Description | The menu contains a sub menu where we can filter the data. We can filter data using and, or, not. Also we can sort data that starts with a specific letter\digit, ends with a specific letter\digit, and contains a specific letter\digit. And user can sort data using the algorithm of his choice. |
|  |  |
| Validators | None |

**UI Component Details**

|  |  |  |
| --- | --- | --- |
| UI Component Name | Type of UI component | Purpose of UI Component/Other details |
| Stop | Button | To stop data scrapping |
| Resume | Button | To Resume data scrapping |
| Play Button | Button | To start data scrapping |
| Filter | Button | To find results |
| Clear | Button | To clear the text boxes |
| Table | Table | To display the data |
| Text Boxes | Text Box | To enter our searches |
| Search Boxes | Text Box | To enter our Searches |
| Export CSV file | Button | To export file into excel |

# **Project Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Use Case Id** | **Use Case Name** | **Member Name** | **Estimated Completion Date** |
| U01 | Main Screen | Khadija & Momina | 03/11/21 |
| U02 | Next Screen | Khadija & Momina | 03/11/21 |
| U03 | Drop Down | Khadija & Momina | 03/11/21 |
| U04 | Filter | Khadija & Momina | 03/11/21 |