**SOUND HUT**

CS162- Final Project Requirement Document



**Project Supervisor**

Mr. Samyan Qayyum Wahla

**Project Members (G53)**

|  |  |
| --- | --- |
| Fatima Mujahid | 2020-CS-138 |
| Malaika Munawar | 2020-CS-112 |

Department of Computer Science

University of Engineering and Technology, Lahore

Table of Contents

[1 Project Detail 4](#_Toc87030195)

[1.1 Executive Summary 4](#_Toc87030196)

[1.2 Business Case 4](#_Toc87030197)

[1.2.1 Outline the business need for the project 4](#_Toc87030198)

[1.2.2 End user of the product 5](#_Toc87030199)

[1.2.3 Motivation for Project 5](#_Toc87030200)

[1.2.4 Level of Impact 5](#_Toc87030201)

[2 Technical Details 6](#_Toc87030202)

[2.1 Attributes Detail 6](#_Toc87030203)

[2.1.1 Name of Entities 6](#_Toc87030204)

[2.1.2 Attributes of Entity 6](#_Toc87030205)

[2.2 Sample of Scrapping Source 7](#_Toc87030206)

[2.3 GitHub Repository Link 7](#_Toc87030207)

[2.4 Environments Detail 7](#_Toc87030208)

[3 Algorithm Detail 8](#_Toc87030209)

[3.1 Sorting Algorithms 8](#_Toc87030210)

[Selection Sort 8](#_Toc87030211)

[Insertion Sort 11](#_Toc87030212)

[Merge Sort 14](#_Toc87030213)

[Shell Sort 18](#_Toc87030214)

[Heap Sort 20](#_Toc87030215)

[Quick Sort 23](#_Toc87030216)

[Counting Sort 27](#_Toc87030217)

[Bubble Sort 29](#_Toc87030218)

[Bucket Sort 31](#_Toc87030219)

[3.2 Searching Algorithm 34](#_Toc87030220)

[Linear Search 34](#_Toc87030221)

[Binary Search 36](#_Toc87030222)

[Exponential Search 37](#_Toc87030223)

[Interpolation Search 39](#_Toc87030224)

[4 Achievement & Integration 40](#_Toc87030225)

[4.1 Integration 40](#_Toc87030226)

[4.1.1 Source of Integration 40](#_Toc87030227)

[4.1.2 Complications Faced 42](#_Toc87030228)

[4.1.3 Final Chosen Website 43](#_Toc87030229)

[4.2 CSV File 43](#_Toc87030230)

[4.3 UI Implementation 44](#_Toc87030231)

[4.3.1 Qt Designer 44](#_Toc87030232)

[4.3.2 Pencil Tool 45](#_Toc87030233)

[4.4 Collaboration 47](#_Toc87030234)

[5 Code Execution 47](#_Toc87030235)

[5.1 Converting to Python 47](#_Toc87030236)

[5.2 QTable Widget 47](#_Toc87030237)

[5.3 Mouse Events 48](#_Toc87030238)

[5.4 Incomplete ideas: 49](#_Toc87030239)

[5.4.1 Progress Bar: 49](#_Toc87030240)

[5.4.2 Multi-Level Sorting: 50](#_Toc87030241)

[6 User Manual 50](#_Toc87030242)

[6.1 Start Scrapping 50](#_Toc87030243)

[6.2 Sorting a Data 51](#_Toc87030244)

[6.3 Searching a Data 52](#_Toc87030245)

[6.4 Viewing a Data 53](#_Toc87030246)

[7 Task Division 53](#_Toc87030247)

# Project Detail

## Executive Summary

Our system will scrap the data of one million songs from multiple websites like Sound cloud, Sound click etc. In Web Scraping it will scrape data with attributes such as Song name. Its Artist, Genre, Sub-Genre, Album its price and all the tags it contain.

Our System will also help in arranging data of Songs. Users will be given a variety of Data in Sorting of Songs. He/She can sort Data according to the Name of a song, Name of a Singer, Album Name, Genre, Sub-Genre and Price. Users can arrange data in ascending or descending order or row and column wise as per his/her choice. Multiple algorithms will be used to sort data. Time analysis of each algorithm will be given below. After Sorting User will be provided with a time taken for sorting. That how much time is taken by each algorithm to sort data so we can detect that which algorithm is performing best. User also have an option of Start, Pause and stop on in progress bar. It will give us percentage that how many data is loaded and we can also pause and stop the loading of data. We will also provide a Searching option for user convenience. Users can search in any column he/she wants. We will also use multiple algorithms in searching

Basically, our program is providing these features sort the data using a variety of sorting algorithms sort the data in a variety of ways (Row or Column), time analysis after Sorting a Column, performing a search based on each column, sorting and searching on several levels

In addition, to make it more effective, a legitimate graphical user interface (GUI) will be introduced in which the client will have the accompanying features:

* Sort the Data According to multiple Sorting Algorithms
* Sort the Data in Multiple Ways (Row or Column Wise)
* Time Details after Sorting a Column
* Searching based on Each Column
* Multi-level sorting and searching

## Business Case

### Outline the business need for the project

Obviously, it's a period when music is a gigantic wellspring of diversion and loads of individuals pay attention to music however its exceptionally difficult to come by all the music as indicated by your desire for a solitary application or site so we can scratch information from different sites which can additionally be use in numerous melodies applications and client can without much of a stretch track down their main tunes from their number one artists

### End user of the product

Web scratching is very valuable since it permits fast and proficient extraction of information from various sources. But let us talk about our top customers which will benefit from this:

* **App Developer**

Individuals who fill in as a developer and are making songs applications like Sound cloud and so forth will be get helped when they will make application they need all data of songs so our information will give enormous amount of help by giving every one of the subtleties of songs

* **Videographers**

This undertaking can likewise help Videographers who need a wide assortment of tunes for their videos. This information will give bunches of data so they can discover tune from their #1 film or vocalist without any problem

* **Public**

This data is also useful for everyday public who are interested in songs. They can find songs with the song name, singer, its movie name or album by search algorithm.

### Motivation for Project

As we probably are aware, it's a time in which Music is a colossal wellspring of amusement and assumes significant parts in engaging our life. Everyone has their own music taste so it's exceptionally difficult to mastermind music arranged by audience decision.

It is Hard to choose whether client like hip hop music, rap melodies and so forth .Thus, our Sound Hut will provide incredible assistance in orchestrating music as indicated by client taste however for this we need wide assortment of music so there are a lot of choices for client to pick music as per his taste

To overcome these issues we will accumulate an enormous assortment of information about melodies from numerous sites and applications. In collecting large amounts of Data web scraping is one of the best option, it gathers and designs information a lot quicker which will give us incredible assistance

### Level of Impact

* **Impact if Project proceeds:**

This assistance could be very proficient and an incentive for various companies. If Project continues it will be useful for music application designers they will depend on it for tune information. It will help us in online marketing as well.

* **Implications if Project not proceeds:**

As we probably are aware, Data scratching has different utilizations and it is given by many organizations. Thus, there will be no ramifications if project doesn't continue except for on the off chance that we make it more efficient and provide innovative features it will definitely get reach however in the event that it doesn't proceed and we show negligence those association will wind up with no boldness accept on us

# Technical Details

## Attributes Detail

### Name of Entities

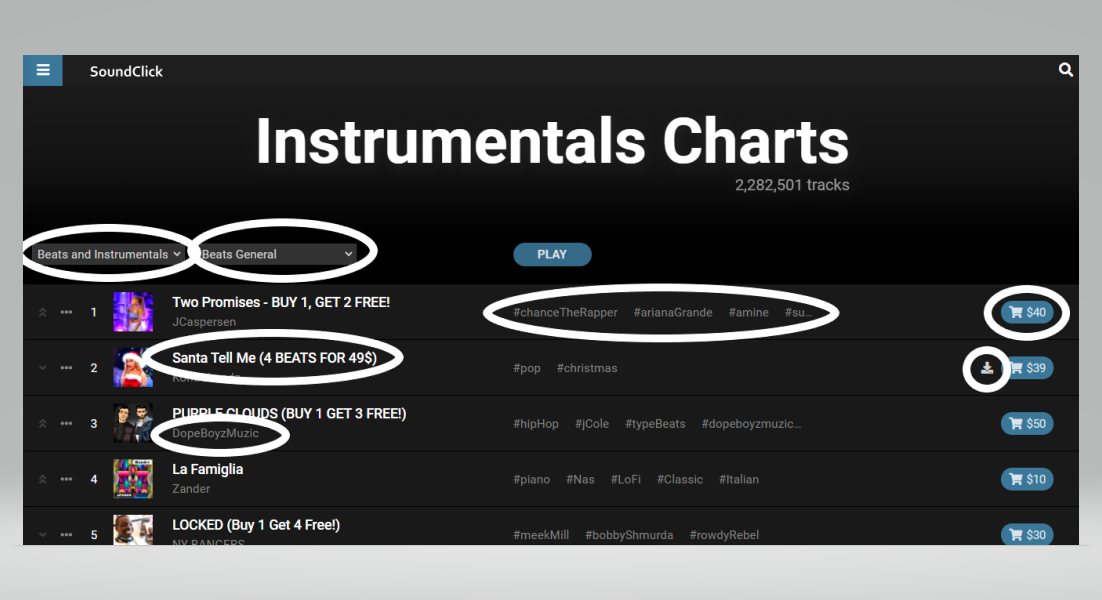
According to the requirement, there ought to be at least 6 entities which are needed for the project. Up Till now, I have chosen some of them which are given below:

* Song Title
* Genre
* Sub-Genre
* Artist Name
* Album Price
* Tags

### Attributes of Entity

|  |  |  |
| --- | --- | --- |
| Name | Data Type | Description |
| Song Title | String | This attribute is basically title or name of a song |
| Artist Name | String | Tells the name of person who sang the song |
| Genre | String | Music is divided into sub-categories or genres depending on the musical instruments. So this attribute will save the genre of music. |
| Sub-Genre | Date | Each genre has a sub-genre. This category will define it. |
| Album Price | String | This category will tell the price of album or if it’s free or not. |
| Tags | String | This category will contain all tags of songs which will help user in searching of song of similar tags. |

## Sample of Scrapping Source



## GitHub Repository Link

<https://github.com/fatymaah31/CS261F21PID53>

## Environments Detail

|  |  |
| --- | --- |
| IDE | Jupyter |
| Language | Python |
| Additional Softwares | Qt Designer, MS Word, MS Excel, BeautifulSoup, Chrome Web Driver |

# Algorithm Detail

## Sorting Algorithms

|  |  |
| --- | --- |
| Selection Sort |  |
| Description | Selection Sort is one of the simplest sorting algorithms. In selection sort, the least value among the array's unsorted items is chosen in each pass and placed into the appropriate place. It's a sorting method that uses in-place comparisons.  The array is separated into two halves in this algorithm: the first is sorted, while the second is unsorted. The sorted component of the array is initially empty, while the unsorted section contains the specified array. The sorted items are on the left, while the unsorted items are on the right. Then, the first smallest element in an unsorted array is chosen and placed at the top. The second smallest element is then chosen and positioned in the second location. The process repeats until we get the fully sorted array.  It is known as Selection sort because it continuously chooses the next-smallest element and swaps it into position. It is normally used when a small array has to be sorted, there is a compulsion to check all elements and swapping cost doesn't matter. Its designing method is an example of Greedy approach and Divide & Conquer. In each iteration of selection sort, the unsorted subarray's minimal element (in ascending order) is chosen and transferred to the sorted subarray. Clearly, sorting the array in this manner is a greedy technique.  Selection Sort is basically an algorithm that works by repeatedly finding the smallest element from an unsorted part in ascending order and placing it in its final location. The Selection sort method works by identifying the least or largest element in a sorted list and then placing it in the right location in a sorted array. Selection Sort is a non-stable sorting Algorithm The minimal element is found and then swapped with the element in the location of the minimum element to put it in its proper place; since it swaps non-adjacent elements it is a non-stable. |
| Pseudo Code | size = length. A  for i=1 to (size-)1:  min=i  for j=i+1 to size:  if(A[j]<A[min]):  min=j  End if  End for  A[i],A[min]=A[min],A[i]  End for  return A |
| Code in Python | **def Selection\_Sort(A):**  size= len(A)  for i in range(size-1):  min=i  for j in range(i+1,size):  if(A[j]<A[min]):  min=j    A[i],A[min]=A[min],A[i]  return A |
| Time Complexity Analysis | * **Best Case:-**   O(n²)   * **Average Case:-**   O(n²)   * **Worst Case:-**   O(n²) |
| Proof of Correctness | Following are the steps for to Proof the Correctness:-   * **Initialization:-**   Start a selection sort loop that goes over each index in the array. Prior to the first iteration of the loop, we choose one element from the array and consider it to be the smallest element in the whole array for the outer loop. And if we suppose that the whole array is made up of a single element, the array is already sorted, at least for the first index.   * **Maintenance:-**   Now we'll demonstrate that loop variation is preserved for each iteration of the loop. The body of the loop will execute n times since the initial use of min has to search for every element in the array. Total number of elements used in the array are denoted by n. The initial elements are now sorted, and the loop will run from 1 to n indexes in the array, switching elements as needed. The loop will run for the next iteration, by checking elements n-1 to n.   * **Termination:-**   The loop iteration = n when the loop is terminated. After comparing the array of smallest elements with minimal we will get a sorted array. This will result in a sorted array and the loop will be terminated. The algorithm is demonstrated to be valid by going over each iteration of the loop invariant. |
| Three Strengths | * No additional storage required * Performs well on small datasets * Uses few operation so more economical |
| Three Weakness | * Poor Efficiency in case of large datasets * Not a stable algorithm * The worst-case scenario is disastrous. |
| Dry Run | https://lh5.googleusercontent.com/Ti59j0sWTXFc2vYCoRJvWv797X20crA3tczTfhUpzH652mTx49l32Y3EiEAuBHKG9oBB3NxqIGeX_m2owf3UmweQ8ZduqpjHbOeCkddfxyBYK6zzfGlNN9HqqN2Cggh1dAVhh9Ck |

|  |  |
| --- | --- |
| Insertion Sort |  |
| Description | It is a sorting algorithm that places an unsorted component at its suitable spot in each iteration. Even though the array is unsorted, the initial element in the array is considered sorted in an insertion sort. Each entry in the array is compared to the previous elements in an insertion sort, resulting in a developing sorted output list. The sorting algorithm removes one element at a time, finds the right spot inside the sorted array, and puts it there with each iteration. The process is repeated until the entire list has been sorted. Insertion sort iterates over the input list, eating one element each time, and producing a sorted output list. Insertion sort takes single element from the source data at a time, identifies where it belongs in the sorted array, and places it there. It keeps on until there are no more input items.  It works similarly as a sorting card in a card game. We choose an unsorted card since we presume the first card is already sorted. The unsorted card moves to the right if it is greater than the one in hand; otherwise, it goes to the left. Other unsorted cards are taken and arranged in their right order in a similar manner. The similar strategy is used in Insertion Sort.  The insertion sort is used in the following situations mostly when there are just a few elements remaining to sort in the array or there are few elements that need to be sorted. Another advantage of the insertion sort is that it just requires a fixed amount of memory for the entire process. It outperforms other related algorithms like bubble sort and selection sort. In other words, with the movement of higher-ranked items, an insertion sort supports in the formation of the final sorted list, one element at a time. The advantages of an insertion sort are its simplicity and low overhead. It's a stable sorting algorithm because the relative order of elements is not modified during insertion sort. |
| Pseudo Code | for j = 2 to A.length  Key = A[j]  i = j - 1  while i>0 and A[i] > Key  A[i+1] = A[i]  i=i-1  End While  A[i+1] = Key  End for |
| Code in Python | **def Insertion\_Sort(A):**  for j in range (2 , A.length ):  key = A[j]  i = j - 1  while i>0 and A[i] > key  A[i+1] = A[i]  i=i-1  A[i+1] = key  return A |
| Time Complexity Analysis | * **Best Case:-**   O(n)   * **Average Case:-**   O(n²)   * **Worst Case:-**   O(n²) |
| Proof of Correctness | Following are the steps for to Proof the Correctness:-   * **Initialization:-**   In fact, loop invariant is always true for the initial element before the loop iteration begins. Before the first repetition, as can be seen from the pseudo code, j is starting from 2. The sorted array would only contain one element A[1] at this time, because an array with one element is presumed to be sorted. As a result, loop invariant holds before the first loop iteration.   * **Maintenance:-**   Now I'll attempt to demonstrate that each array element retains the loop invariant. As previously stated, the for loop's body shifts elements to the right side and inserts the necessary element in the appropriate element, such as A[j-1], A[j-2], and so on. The sorted elements are then grouped into a sub-array. To keep and maintain the flow of the for loop for the following iteration, we increase then j. The outer and inner loops are therefore entirely preserved.   * **Termination**   Finally, we look at what occurs when the cycle comes to a conclusion. j in range (2, A.length) is the condition that causes the for loop to finish. We should have j = n + 1 around then, since each loop cycle grows j by 1. Substituting n + 1 for j in the wording of loop invariant, we get the subarray A [1...n], which consists of the components from A [1...n] but in sorted order. We assume that the whole show is ordered since the subarray A [1...n] is the entire exhibit. As a result, the computation is correct. |
| Three Strengths | * Suitable for smaller datasets * Minimal usage of memory * Stable sort |
| Three Weakness | * Not suitable for larger datasets * Large number of shifting required * Efficient for smaller data only |
| Dry Run |  |

|  |  |
| --- | --- |
| Merge Sort |  |
| Description | Merge Sort is the most efficient algorithm. It is based on the divide-and-conquer strategy. Merge sort divides a list into sub lists until each has just one element, then merges them into a sorted list.  In Merge Sort Algorithm we have two approaches :   1. Top-Down 2. Bottom-Up   The recursion technique is used in the top-down merge sort process. It starts at the top and works its way down, with each recursive turn asking the same question: what is necessary to sort the array? with the response being to divide the array into two parts, use a recursive function and merge the results.  Iterative process is used in the Bottom-Up merge sort strategy. It starts with a "single-element" array and merges two neighboring elements while sorting them simultaneously. The combined sorted arrays are merged and sorted again until only one single unit of sorted array remains.  Merge sort continuously cuts down a list into numerous sub lists until each sub list has only one entry, then merges those sub lists into a sorted list. Merge sort splits the list into equal halves until there is no more room for division. If there is just one entry in the list, it is sorted by definition. We use the merge() function to get our final sorted array by combining the sub-array. As the same element in an array maintains its original position in relation to other elements in the array. So, Merge Sort is a stable sort. In Merge Sort, the mass of implementations provide a stable sort. |
| Pseudo Code | **MergeSort(arr, p, r):**  if p < r  q = [(p + r)/2 ]  mergeSort(arr, left, mid)  mergeSort(arr, mid+1, right)  merge(arr, left, mid, right)  end |
| Code in Python | **def Merge(Arr1, Arr2, Array):**  i=0  j=0  k=0  while (i < len(Arr1) and j < len(Arr2)):  if Arr1[i] < Arr2[j]:  Array[k] = Arr1[i]  i=i+1  k=k+  else:  Array[k] = Arr2[j]  j=j+1  k=k+1    for i in range (i, len(Arr1)):  Array[k] = Arr1[i]  i=i+1  k=k+1  for j in range (j,len(Arr2)):  Array[k] = Arr2[j]  j=j+1  k=k+1  return Array  **def MergeSort(Array,num):**  num2=num%2  if num2 == 0:  mid=num//2  else:  num2=num%2  mid=num//2  mid=num2+ mid  Arr1=[]  Arr2=[]  for i in range(mid):  temp=Array[i]  Arr1.append(temp)  count=0  for i in range(mid,num):  temp=Array[i]  Arr2.append(temp)  count=count+1  if(num>mid):  MergeSort(Arr1,mid)  MergeSort(Arr2,num-mid)  Merge(Arr1,Arr2,Array)  return Array |
| Time Complexity Analysis | * **Best Case:-**   O(nlogn)   * **Average Case:-**   O(nlogn)   * **Worst Case:-**   O(nlogn) |
| Proof of Correctness | Following are the steps for to Proof the Correctness:-   * **Initialization:-**   Before the initial iteration of loop, As k is equal to p so sub-array A[p.. k-1] contains nothing. Empty subarray with smallest elements of L and R contains k-p = 0. As we can see both i and j are equal to 1. So, L[i] and R[j] are the smallest elements in their arrays that haven't been transferred back into our actual sorted array A.   * **Maintenance:-**   Assume Merge meets the loop invariant condition up to k. L and R's smallest elements (k-p+1) are already sorted in A.The lowest number left in L and R is the next one to be placed. This number is much higher than the preceding ones in A. For k+1, the loop invariant condition is fulfilled.   * **Termination:-**   Merge terminates when k becomes equal to r + 1 such that when r-p+1 elements have been entered into A and all elements have been sorted. The algorithm is demonstrated to be valid by going over each iteration of the loop invariant. |
| Three Strengths | * More efficient * Work well for all type of datasets * Stability |
| Three Weakness | * Additional memory space required * Its running time is 0(n log n). * Poor locality of reference |

|  |  |
| --- | --- |
| Shell Sort |  |
| Description | Shell Sort was created with the intention of allowing exchange of far-flung items. The insertion sort algorithm is an extended variant of shell sort. It starts by sorting items that are far away, then gradually minimizes the distance between the components to be sorted. Depending on the sequence, the interval between elements is reduced. The shell sort improves on the insertion sort by splitting the original list into smaller sub lists, each of which is sorted using an insertion sort. The key to the shell sort is the unique method these sub lists are selected for. Rather than breaking the list into sub lists of contiguous components, the shell sort usually employs an increment of i to generate a sub list by choosing parts that are i elements apart, which is termed gap.  Shell sort enhances the efficiency of insertion sort by rapidly transferring values to their intended location. The insertion sort method is a more advanced version of the shell sort algorithm. It begins by sorting far-away things, then progressively reduces the distance between the components to be sorted. The spacing between items is decreased depending on the sequence. The shell sort outperforms the insertion sort by separating the original list into smaller sub lists, each of which is sorted using an insertion sort. The shell sort's important feature is the one-of-a-kind process by which these sub lists are chosen.  Shell Sort uses multiple sequences and its performance is decided by the sequence used in it. As the same element in an array does not maintain its original position in relation to other elements in the array. So, Shell Sort is not a stable sort. Usually insertion sort fails to perform well when close elements are far apart then Shell sort is used as it helps in shortening the distance between adjacent items. As a result, the number of swaps required will be reduced. |
| Pseudo Code | **ShellSort( A )**  m= len (A) // 2  While m > 0  i=0  J= m  While j < len(A)  If A[i] > A[j]  Swap A[i] and A[j]  i++  j++  k=1  while k – m> -1  if A[k-m] > A[k]  Swap A[k-m] and A[k]  K-- |
| Time Complexity Analysis | * **Best Case:-**   O(nlogn)   * **Average Case:-**   O(nlogn)   * **Worst Case:-**   O(n²) |
| Proof of Correctness | Following are the steps for to Proof the Correctness:-   * **Initialization:-**   The shell sort's setup is identical to that of the insertion sort. The shell sort's initial loop is j len(A), which swaps the previously settled positions with their partial gaps. As a result, for the first settled index the first element would be sorted according to the unbiased gap. So, loop invariant holds before the first loop iteration.   * **Maintenance:-**   Now I'll attempt to demonstrate that each array element retains the loop invariant. The loop's body iterates in such a manner that it shifts elements to the right side and inserts the necessary element in the proper location, such as A [j-1], A [j-2], and so on. The sorted elements are then grouped into a sub-array. To keep and maintain the loop's flow for the following iteration, we increase then j. The outer and inner loops are therefore entirely preserved. And is swapped when it is deemed acceptable.   * **Termination**   Finally, we look at what occurs when the cycle comes to a conclusion. j in range (2, A.length) is the condition that causes the for loop to finish. We should have j = n + 1 around then, since each loop cycle grows j by 1. Substituting n + 1 for j in the wording of loop invariant, we get the subarray A [1...n], which consists of the components from A [1...n] but in sorted order. We assume that the whole show is ordered since the subarray A [1...n] is the entire exhibit. As a result, the computation is correct. |
| Three Strengths | * Efficient for finite data only * Faster than bubble sort * Perform better in medium sized data |
| Three Weakness | * Complex Algorithm * Slower than Quick, Merge and Heap Sort * Unstable Sort |

|  |  |
| --- | --- |
| Heap Sort |  |
| Description | A sorting algorithm that begins by organizing the data to be sorted into a heap, a kind of binary tree. Heap sort is a sorting algorithm for putting a list of elements in order. The Heapsort algorithm employs the Heap Tree, a tree concept. It uses Heap instead of Node.  In heap Sort Algorithm there are two ways to arrange list of elements:   1. Max Heap 2. Min Heap   To arrange the list of elements in descending order we use max heap and min heap is used to arrange elements in descending order. The Heap Sort algorithm performs well when we get sorted data in a binary tree. In Heap Sort we use comparison-sorting technique to get an array sorted. It's comparable to selection sort, in which we discover the maximum element first and then put it at the end. The method is repeated for the last element. Because actions in the heap might modify the relative order of comparable keys, heap sort is not stable. To save space and resources, the binary heap may be represented using array-based approaches. Heap sort is an in-place algorithm that overwrites inputs without needing any additional data structures at runtime. While other arranging calculations might develop dramatically slower as the quantity of things to sort increment, the time needed to perform Pile sort increments logarithmically. This recommends that Stack sort is especially appropriate for arranging an immense rundown of things. |
| Pseudo Code | **function MAX-HEAPIFY (A,i)**  l = LEFT (i)  r = RIGHT(i)  if (l <= A.heap-size and A[l] > A[i])  largest = l  else largest = i  if r <= A.heap-size and A[r] > A[largest]  largest = r  if largest != i  exchange A[i] with A[largest]  MAX-HEAPIFY(A, largest)  **function BUILD-MAX-HEAP(A)**  A:heap-size =A.length  for i =A.length/2 downto 1  MAX-HEAPIFY(A)  **function HEAPSORT(A):**  BUILD-MAX-HEAP.(A)  for i = A.length downto 2  Swap A[1] with A[i]  A.heap-size =A.heap-size -1  MAX-HEAPIFY.(A,1) |
| Code in Python | **def MAX\_HEAPIFY(A, size, i):**  l = 2\*i  r = 2\*i+1  if (l <= size and A[l] > A[i]):  largest = l  else:  largest = i    if (r <= size and A[r] > A[largest]):  largest = r    if (largest != i):  A[i],A[largest]=A[largest],A[i]  MAX\_HEAPIFY(A,size,largest)  **def BUILD\_MAX\_HEAP(A):**  n=len(A)  for i in range(n//2-1 , -1 ,-1):  MAX\_HEAPIFY(A,n,i)    **def HEAPSORT(A):**  BUILD\_MAX\_HEAP(A)    for i in range(len(A)-1, 0 , -1):  A[0],A[i]=A[i],A[0]  MAX\_HEAPIFY(A,i,0) |
| Time Complexity Analysis | * **Best Case:-**   O(nlogn)   * **Average Case:-**   O(nlogn)   * **Worst Case:-**   O(nlogn) |
| Proof of Correctness | Following are the steps for to Proof the Correctness:-   * **Initialization:-**   Before the first loop iteration, i = [n/2]. Each node is a leaf and hence the root of a trivial max-heap [n/2] + 1; [n/2] + 2,..., n.   * **Maintenance:-**   Observe that the offspring of node i are numbered higher than i to demonstrate that each iteration keeps the loop invariant. As a result of the loop invariant, they are both roots of max-heaps. This is the requirement that the call MAX-HEAPIFY (A, i ) must meet in order to make node i a max-heap root. Furthermore, the MAX-HEAPIFY function keeps the property that nodes i+1; i + 2,...., n are all max-heaps roots. The loop invariant is reestablished for the following iteration by decreasing i in the for loop update..   * **Termination**   When the program ends, i=0. Each node 1, 2,..., n is the root of a max-heap according to the loop invariant. Node 1, specifically |
| Three Strengths | * Flexibility * Stable sorting algorithm * Fast for large sets of data |
| Three Weakness | * Time-Taking * Management of memory is complex * Not a stable sort |

|  |  |
| --- | --- |
| Quick Sort |  |
| Description | Quick Sort is quite similar to Merge Sort. It follows the paradigm of Divide-and-Conquer. But its whole working depends on the Pivot. We partition the array in two parts after choosing the pivot. We can choose pivot in four different ways. Pivot can be the first, median or last element or we can randomly choose it.  Quicksort's principle is to continually separate the "large" elements from the "small" elements. The algorithm's initial step is to choose a "pivot" which will help us in splitting large and small values. Each quicksort implementation has its own technique for determining the pivot value, some of which are much superior than others. The pivot value in the implementation below is just the first element of the list. All values lower than the pivot are put at the beginning of the set, and all values bigger than the pivot are moved to the right after the pivot value has been chosen. This procedure effectively places the pivot value in the proper location every time. After that, each side of the pivot is quick sorted.  Quick Sort is named Quick just because it has the ability to sort Array 2 to 3 times faster than other sorting algorithms but we prefer a short inner loop for it. One of its benefits is that it can sort large sets of data efficiently without requiring any additional space. But it's an unstable algorithm. |
| Pseudo Code | **QuickSort**  If low<high  p = partition(array, low , high)  QuickSort(array , low, pi-1 )  QuickSort(array , pi+1, high ) |
| Code in Python | **def partition(array, low, high):**  i= low - 1  Pivot =array[high]  For j in range(low, high):  If (array[j]<=pivot):  i=i+1  array[i] ,array[j]= array[j] , array[i]  **def QuickSort(array, low , high):**  if(low<high):  p = partition(array, low , high)  QuickSort(array , low, pi-1 )  QuickSort(array , pi+1, high ) |
| Proof of Correctness | Following are the steps for to Proof the Correctness:-   * **Initialization:-**   Prior to the first loop iteration, I = p-1 and j =p. The first two criteria of the loop invariant are trivially met since no values exist between p and I and no values exist between I +1 and j -1. The third criterion is satisfied by the assignment in line   * **Maintenance:-**   Depending on the result of the test in line 4, we examine two scenarios.   1. It demonstrates what occurs when A[j] > x; the loop's only activity is to increase j. Condition 2 holds for A[j] - 1 after j is increased, while all other entries stay unaltered. 2. The loop increases I swaps A[i] and A[j], and then advances j when A[j] is less than or equal to x. We now have A[i] smaller than or equal to x as a result of the swap, and condition 1 is met. Similarly, A[j-1] > x, since the item swapped into A[j -1] is more than x according to the loop invariant.  * **Termination:-**   When the program ends j=r. As a result, every element in the array belongs to one of the invariant's three sets, and we've divided the array's values into three groups: those less than or equal to x, those bigger than x, and a singleton set containing x**.** |
| Three Strengths | * Additional storage not required * Inner Loop is extremely short * Efficient |
| Three Weakness | * Extremely Complicated if not recursive * Quadratic time in worst case * Unstable |

|  |  |
| --- | --- |
| Counting Sort |  |
| Description | Counting sort Algorithm is used to sort a collection of objects or elements with the help of keys that are integers basically. The key is set and then it counts the number of elements with that key. After that, it performs some arithmetic operations to determine the position of each object in the array or list. It is used in situations when the variation in keys is not very much larger than the number of values. In the Counting sort Algorithm, we face memory allocation issues quite often. If we compare it with Bucket Sort, then it will store a single number for each bucket. Counting sort is not a comparison type of sort .The key is set first and then the key is used to determine the indexes. Counting sort is a stable sort. Stability of an algorithm matters when we are having multiple values with the same keys.  Time Complexity of counting sort is O(n+k), here n represents the size of the array and k denotes the number of keys. Counting Sort might not be so efficient if the key values are very large i.e. if k is very large. |
| Pseudo Code | **CountingSort(A)**  Lett C[0 : : k] be a new array  for i = 0 to k  c[i] = 0  for j=0 to n  c[A[j]] = c[A[j]] + 1  for i=1 to k  c[i] = c[i] + c[i-1]  for j= n-1 down to 0  B[c[A[j]]-1] = A[j]  c[A[j]] = c[A[j]]-1 |
| Code in Python | **def CountingSort(inpu):**  k = 10  count=[]  output =[0]\*len(inpu)  for i in range (0 , k):  count.append(0)    for i in range(0, len(inpu)):  count[inpu[i]] = count[inpu[i]]+1    for i in range(1 ,k):  count[i] =count[i]+ count[i - 1]    for i in range(len(inpu)-1,-1,-1):  output[count[inpu[i]]-1]=inpu[i]  count[inpu[i]]=count[inpu[i]]-1    return output  inpu=[4, 3 ,2, 9 ,4 ,1 ,0 ,8,5 ,7,5 ]  output=CountingSort(inpu)  print("Output : "+str(output)) |
| Time Complexity Analysis | * **Best Case:-**   O(n+k)   * **Average Case:-**   O(n+k)   * **Worst Case:-**   O(n+k) |
| Proof of Correctness | Following are the steps for to Proof the Correctness:-   * **Initialization:-**   We initialize the array A that needs to be sorted.   * **Maintenance:-**   The keys are formed and then the elements with similar keys are found. Then they are sorted to form a sorted array   * **Termination**   The loop terminates when all the elements get in the right order. |
| Three Strengths | * Linear Time complexity * Reduced space complexity * Good for educational purposes i.e. can be used as an introductory sorting algorithm |
| Three Weakness | * Counting sort works only for discrete values such as integers * Restricted Inputs * Space Cost |

|  |  |
| --- | --- |
| Bubble Sort |  |
| Description | Bubble sort is all about swapping. It compares elements and swaps them if the order is not correct and the swapping continues till the array or list gets sorted. The worst case of bubble sort has a time complexity of O(n2) where n means the size of the array to be sorted. Practically, bubble sort is not a sorting algorithm because the other sorting algorithms we have are not complex and they are way faster than bubble sort. For Example, Insertion sort has a worst case time complexity of O(nlogn).  So as we are comparing bubble sort with other algorithms, we must talk about the advantages it has over other sorting algorithms. The only advantage that bubble sort has over other sorting algorithms (excluding Insertion Sort), is that it has a kind of algorithm that detects the status of the list, that is, if the list is sorted or not and this function is already built in the algorithm. When the list is already sorted, we call it “Best case”. The time complexity of the base case for bubble sort is O(n). From this we can undoubtedly say that the best case time complexity of bubble sort is better than all the sorting algorithms (except Insertion Sort, it has the same advantage over other sorting algorithms). Moreover, Bubble Sort will still do better if the list is partially sorted.  In today’s world, bubble sort is of no use for we have highly efficient sorting algorithms but still, bubble sort is used for educational purposes, that means it is used as an introductory sorting algorithm to computer science students like us due to its simplicity and the fact that it’s concept is easy to grasp. The problem we face in bubble sort is that the greatest element goes to the end of the list after each iteration, that means the sorted data gets sorted again and again each time the loop executes, therefore, we restrict the inner loop to avoid this. |
| Pseudo Code | For i = 1 to A.length-1  For A.length down to i+1  If A[j] < A[j-1]  Swap A[j] and A[j-1] |
| Code in Python | **def BubbleSort (array , size):**  For i in range (size-1):  For j in range (0, size -1):  If arr[j] >arr[j+1]:  Arr[j] , arr[j+1] = arr[j+1] , arr[j] |
| Time Complexity Analysis | * **Best Case:-**   O(n)   * **Average Case:-**   O(n²)   * **Worst Case:-**   O(n²) |
| Proof of Correctness | Following are the steps for to Proof the Correctness:-   * **Initialization:-**   Initializing the sub array A [1 …. I-1] and this is the smallest element in the array.   * **Maintenance:-**   After initializing, A[i] will be the smallest element of the sub array A[i….n]. And when the outer loop begins to execute, A[1….i-1] contains elements that are smaller than the elements of the array A[i...n]. So after the execution A[1...i] will consist of elements smaller than the values of indexes of A[i+1...n]   * **Termination**   The loop terminates when i= A.length, at this point all the elements have been sorted. |
| Three Strengths | * Require less space * Simply Understandable * Can Easily Implement |
| Three Weakness | * Only work efficient for smaller datasets * Require more time * Extremely slow in some cases |
| Dry Run |  |

|  |  |
| --- | --- |
| Bucket Sort |  |
| Description | Bucket sort works by distributing the data of an array into a number of buckets. The buckets that are formed are then sorted individually by the sorting algorithm of our choice (i.e. we can sort them with any sorting algorithm without any restriction or the bucket sort itself). It is a distribution sort.  If we talk about the code, then we actually create two methods, the first one BucketSort () and the second one will be our desired sorting algorithm, for ease we use Insertion Sort most of the time. The bucket sort time complexity depends a lot on the number of buckets formed.  When the elements of the given array consist of similar keys, then it can cause the algorithm to put more data in one or two buckets and it might create some empty buckets as well. The worst case of this Algorithm is when there is only one bucket where all the elements are in the same bucket. In this case the overall performance of the algorithm depends upon the algorithm used in the code other than bucket sort for example the insertion or the merge sort. For Average case, we assume that the inputs were uniformly distributed. Firstly, we initialize the buckets and find the maximum or the largest values in all the arrays formed.  The time complexity for average case bucket sort will be O(n+k). The bucket sort is really helpful when it comes to memory issues. It is used to sort a large amount of data. Most of the time, it is used when the given data is so huge that it cannot be placed in the memory. And this is because of the fact that all the buckets that are formed, they do not take any auxiliary space in memory. |
| Pseudo Code | Let B[0 . . n-1] be a new array  n=A.length  for i = 0 to n - 1  B[i] an empty list  for i = 1 to n  Insertion of A[i] into list B[[n A[i]]]  For i = 0 to n - 1  Sort list B[i] with Insertion Sort  Concatenate the lists B[0], B[1] , . . . B[n-1] |
| Code in Python | **def bucketSort(size):**  array = []  number = 10  for i in range(number):  array.append([])  for j in size:  b = int(number \* j)  array[b].append(j)  for i in range(number):  array[i] = InsertionSort(array[i])  k = 0  for i in range(number):  for j in range(len(array[i])):  size[k] = array[i][j]  k += 1  return size |
| Time Complexity Analysis | * **Best Case:-**   O(n+k)   * **Average Case:-**   O(n+k)   * **Worst Case:-**   O(n²) |
| Proof of Correctness | Following are the steps for to Proof the Correctness:-   * **Initialization:-**   We initialize the array that is to be sorted let it be A   * **Maintenance:-**   Buckets are formed during the execution and the algorithm is correct if two items present in a bucket have same relative order and if they are in two different buckets they will have same order   * **Termination**   The loop terminates when all of the elements get sorted. |
| Three Strengths | * Quicker to run * Efficient * Stable |
| Three Weakness | * Efficiency depends on the input * Performance is dependent on number of buckets used * If the buckets are not correctly distributed, it requires extra effort. |

## Searching Algorithm

|  |  |
| --- | --- |
| Linear Search |  |
| Description | Linear search is a straightforward and easy search strategy. In linear search, we look for an element or number in an array by traversing it from the beginning until the element or value we want is discovered. It does a one-by-one search of an array or list. A loop is used to go through an array sequentially, beginning with the first element, in linear search. It keeps on comparing each element of the array and comes to a halt when the number or the array's end is reached.  The phases of implementation that will take place are: First of all it uses a for loop to examine the array. In each cycle, compare the goal value to the array's current value. If the values are equal, return the array's current index. Continue to the next array element if the values doesn’t match. -1 will be returned when no same value is found.  For small arrays, the linear search strategy is acceptable, but not for big arrays. The elements in this search algorithm don't have to be placed in any particular sequence. The elements must be organized in a sorted order before the binary search can begin. Any linear data structure, such as an array or a linked list, may be used to do a linear search. Linear search will do quick searches on small to medium-sized lists. Small to larger arrays can be searched quite fast with today's powerful computers. There is no need to arrange the list. Linear searching, unlike binary searching, does not need the use of an ordered list. Insertions and deletions have no effect. Additional entries may be added and removed since the linear search does not need the list to be sorted. Because other algorithms use for searching may have to reorganize the list after insertions or deletions, a linear search may be more efficient in certain cases. But linear search needs more space and its time complexity is more |
| Pseudo Code | **LINEAR\_SEARCH(A, v):**  for i = 1 to A.length  if A[i] == v  return i  return NIL |
| Code in Python | **LINEAR\_SEARCH(A, v):**  for i in range( 1 , len(A))  if (A[i] == v):  return i  return NIL |
| Time Complexity Analysis | * **Best Case:-**   O(1)   * **Worst Case:-**   O(n) |
| Proof of Correctness | * **Initialization**   Before commencement of the first iteration, sub-array A[1…0] is null, and empty array so it doesn’t has v   * **Maintainence**   Assume that subarray A[1...(i1)] does not include v before the i-th iteration. We verify whether A[i]=v on the i-th iteration. If that's the case, we'll print index I and cease looking. If that isn't the case, we know A[i]v and, by assumption, A[j]v where j=1,2,...,i1, so when those two assertions are combined, we get A[j]v where i=1,2,3,...,i. Finally, when we raise i=i+1 at the conclusion of the loop iteration, the invariant will stay true.   * **Termination**   There are two ways that a loop might end. The first is when we locate element v, in which instance we stop and return the index where key v was found. In the other scenario, the loop will end after iterating over all n items. When this occurs, index will be i=n+1, and since subarray A[1...n] does not include v due to the invariant, we will return NIL. |
| Three Strengths | * Search for non-sorted data also * Insertion and deletion doesn’t effect * Fast for small amount of data |
| Three Weakness | * More time complexity * Slow for large data * Search sequentially |

|  |  |
| --- | --- |
| Binary Search |  |
| Description | Binary search is similar as we search word in dictionary. It searches a sorted array by halving the search interval again and over. Begin by creating an interval that spans the whole array. If the search key's value is less than the element in the interval's midpoint, the interval should be narrowed to the bottom half. Otherwise, keep it to the top half of the page. Check the value until it is discovered or the interval is null.  When using binary search on an array, we partition it into distinct search periods in the same way we did with the dictionary. The initial array is the search interval at first. The search interval is the array containing items less than the center item if the value of the requested item is less than the item in the middle of the interval. In the same way, if the sought item is bigger than the item in the center, the search interval is the array of objects larger than the middle item.  Only a sorted list of items may be utilized with the binary search method. That is, the binary search is only utilized with a list of components that has already been organized in a certain order. A list of items organized in random order cannot be searched using the binary search.  One of the key benefits of a binary search is that it is significantly faster than a serial search since the amount of data to be searched is divided in half with each step.  The downside of the BST data structure is that the degree of the tree will constantly rise over time, resulting in poor search efficiency. |
| Time Complexity Analysis | * **Best Case:-**   O(1)   * **Average Case:-**   O(logn)   * **Worst Case:-**   O(logn) |
| Proof of Correctness | * **Initialization:-**   The array is sorted as part of the method's preconditions. Because a.length is at least 1, lr. k is in a[l..r], which is the whole array, and the precondition ensures that k is present.   * **Maintenance:-**   First, note that the array is never altered throughout the loop, preserving component (1) of the invariant. The values of l and r at the conclusion of the loop are represented by l', r'. Then l'r' is required in part (2), and k∈a[l'..r'] is required in part (3). It's worth noting that m is the rounded-down average of l and r. So, lmr, we've established that. Either k∈a[l..m] or k∈a[m+1..r] is known. We look at each scenarios independently.  k∈a[l..m] case: We need k∈a[m] in this situation, so the if guard is true, and r' = m and l' = l. Since lm, we've had l'r' as a requirement. Because k∈a[l..m] is assumed, k∈a[l'..r']. The array has not been changed, therefore it is still sorted.  Situation  k∈a[l..m]: We need r>ml and k∈a[m+1..r] in this case. Because k is beyond the array's necessary range and the array is sorted, the if condition must be false. As a result, l' = m+1 and r' = r. l'>r' as necessary because r>m. We are aware. Because there were no modifications to the array, it is still ordered.   * **Termination:-**.   The invariant (2) ensures that the value r-l is non-negative. We know mr in the situation where ka[l..m], so l'r' lr. In the other situation, we already know l<m+1, thus l'-r' < l-r. Every loop iteration grows smaller since integer division rounds down. As a result, the loop ultimately comes to an end. |
| Three Strengths | * Faster for large arrays * Efficient * Less time consuming |
| Three Weakness | * It can only be used with sorted lists. * Not good for small array * More complexity |

|  |  |
| --- | --- |
| Exponential Search |  |
| Description | Exponential Algorithm works by taking a specific value as input which we refer as key and then searching the key value through the sorted and unbounded list. Exponential search can out-run other search algorithms, even the binary search, if the key element is present in the starting index of the array. The algorithm runs in O(log i) time , where i indicates the index of the element of the list which is to be searched. In unbounded lists, exponential search becomes very useful. This algorithm takes place in two stages. In the first step, it analyzes the key value and determines its range by comparing it to the list elements, that is what would be the range of the key if it were present in the list. After this, in the second stage, binary search is performed on this key.  The worst case time complexity of exponential search is O(log i) , and for average case, it is also the same that is O(log i) , where i denotes the index of the list element. The best case time complexity is given as O(1). |
| Pseudo Code | **EXPONENTIAL\_SEARCH( array, n , x )**  If array[o]==x  Return 0  I=1  While i<n and array[i]<=x  I=i\*2  Return BinarySearch(array , i/2 , min(I,n-1),x) |
| Code in Python | **def Exponential\_Search(array , size , search\_element):**  if array[0] == search\_element:  return 0  i=1  while i<size and arr[i]<=search\_element:  i=i\*2  return binary\_Search(array, i/2,min(i,size-1),search\_element) |
| Time Complexity Analysis | * **Best Case:-**   O(1)   * **Average Case:-**   O(log i)   * **Worst Case:-**   O(log i) |
| Three Strengths | * It is really helpful in unbounded lists * It is an efficient algorithm |
| Three Weakness | * It itself is a search algorithm but it uses binary search. * It works on sorted arrays |

|  |  |
| --- | --- |
| Interpolation Search |  |
| Description | Interpolation Search Algorithm is somewhat similar to the binary search algorithm, where we set a specific value as the target and then search it in the array. In every step, the interpolation search determines the probability that the element to be searched might be present at a particular place in the array by comparing the target or key value to the array elements. The comparison is actually between the “mid” value to find the and the key, that is , we determine the range of the search space and estimated position is compared to the key. Th mid value is calculated using the lowest and highest array values by the formula:  Mid = low +( ( x - A[low] ) \* ( high – low) / (A[high] – A[low]))  The mid position is calculated at each iteration and after that , as in binary search algorithm, it moves in to set a boundary that defines an interval that might contain the target value. |
| Pseudo Code | **INTERPOLATION\_SEARCH( array, size , x )**  low = 0 , mid = - 1 , high = size – 1  While x match not found  If low = high or A[low] = A[high]  Return 0, not found  End if  Mid = low +( ( x - A[low] ) \* ( high – low) / (A[high] – A[low]))  If A[mid] = x  Element found  Else  If A[mid] < x  Low = mid+1  Else if A[mid]>x  High = mid – 1  end |
| Code in Python | **def Interpolation\_Search(array , size , x):**  if (low<=high and x>= arr[low] and x< arr[high])”  Mid = low +( ( x - A[low] ) \* ( high – low) / (A[high] – A[low]))  If arr[mid]==x:  Return mid  If arr[mid]<x:  Return Interpolation\_Search(array, mid+1,high,x)  If arr[mid]>x:  Return Interpolation\_Search(array, mid-1,high,x) |
| Time Complexity Analysis | * **Best Case:-**   O(1)   * **Average Case:-**   O(log2(log2 n))   * **Worst Case:-**   O(n) |
| Three Strengths | * It works better than binary search for unbounded lists. * It is an efficient algorithm |
| Three Weakness | * It itself is a search algorithm but it uses binary search. * It works on sorted arrays |

# Achievement & Integration

## Integration

Song scraping is the practice of extracting material and song data from a website using bots. Here we'll go through how we process our scraping, what websites were used as sources of data for our project, what obstacles we encountered, and why, if any, we changed our source of data. Further, we will discuss how we thought about our project and what consequences we face and how this project is helpful for us

### Source of Integration

In the beginning, we choose several websites which will meet our needs according to project requirement. Basic need of project requirement was that our scraping must include 7 attributes. Our project can exceed the limit of 7 but it should not be less than. Further, we have to scrap data of one million. We have to provide data that must be accurate and helpful in future. Such as, in scraping of songs we had to gather all information about a single song provided on that website. We are compelled to gather data that must be fully understandable and suitable for our project. We have option to gather data from multiple websites.

Following websites were decided for data scraping:-

* Sound click
* Wikipedia

Reference screenshot for scraping of following websites are given below:-

#### Wikipedia

Wikipedia is a free online website that is assisting in the creation of a world where everybody may freely share the total of all knowledge. Here, we have access to number of artists and detail of songs they sang and all the details of it in the year they sang. Album and movie of a song. From here, we can easily access numerous amount of data.

**

Fig 4‑1:Reference snap for song scraping

Following attributes were available on this website:-

* Film
* Song
* Composer
* Writer
* Co-Artist
* Artist
* Year

#### Sound click

Sound Click is a music-centric social network. Songs may be listened to online, downloaded as MP3s, sold in the shop to others. Sound Click creates daily charts for a variety of genres based on unknown variables. It is an international US based website. Here million of data is available which we can scrap.

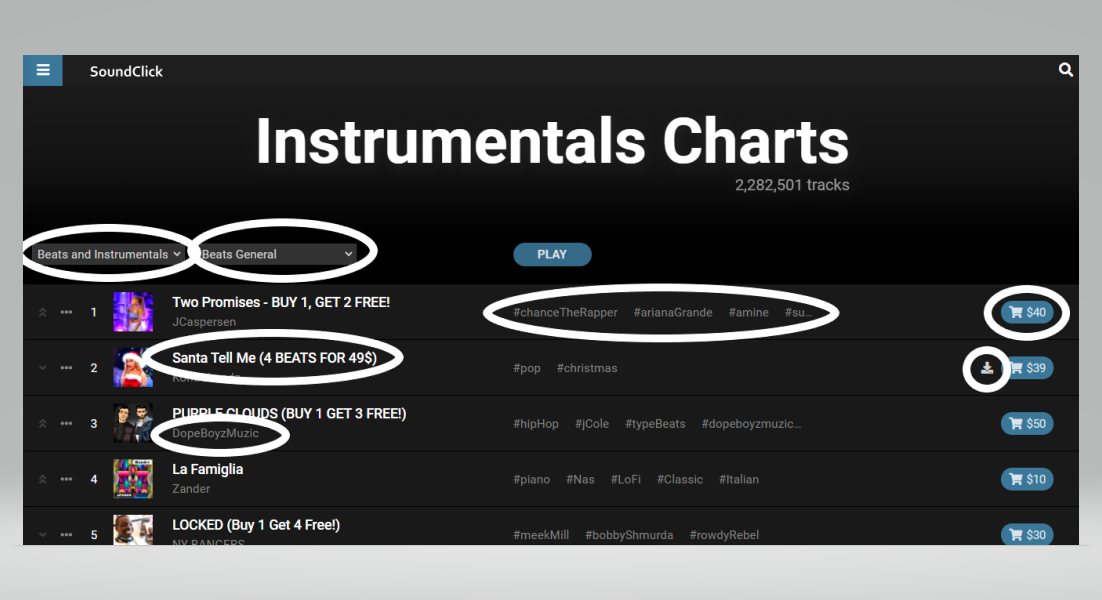


Fig 4‑2Reference snap for song scraping

Following attributes were available on this website:-

* Price
* Song
* Genre
* Sub-Genre
* Tags
* Artist
* Download or Not

### Complications Faced

In process of selecting our final website we face some difficulties. First of all each attributes were matched as they were not similar so we have to choose a single website to scrap our data. Now, question is how we can choose which website is best for our data scraping. To overcome this issue we move forward to our requirements.

#### Reason for choosing single website

Here the question come why we have to choose a single website. First of all attributes on different websites are not same which was not fulfilling our requirement and secondly there were few scraping issues. Many websites don’t allow us scraping which was one of our major issue. Also, solo single website was not providing us one million data so we have to choose a single website with one million data.

### Final Chosen Website

In choosing our final website first we check all of our needs as Sound Click was fulfilling all our requirements so we choose it as our ideal website. We double check that it must allow us scraping of all seven attributes with one million data.

#### Analysis of our Final Website

|  |  |
| --- | --- |
| **Genre** | **Number of tracks** |
| Instrumentals | 2,282,527 tracks |
| Hip-Hop | 1,781,641 |
| Electronic | 339,064 |
| Rock | 268,693 |

Fig 4‑3: Analysis of Final Scraping Website

#### Difficulties during Scraping

In this phase we will explain difficulties we face during scraping:

* **Scraping through Request**

At start we try to get our data through **request** in our scraping data. But due to some reason our website was not allowing us to scrap data. As it was an international website so it was quite hard for us to completely understand the code. Plus website was not allowing us scraping through request.

* **Duplicating Data**

One other issue we face during scraping was our that our data during scraping was duplicating. Each time it scrap data from next page was also scraping from first page. So, our data keep on duplicating which was one of the major issue.

### Overcoming Scraping Problems

Following steps were taken in order to overcome scraping issue

* **Using Web Driver**

When our website was not allowing scraping through request we use google web driver to scrap data from this

* **Duplication resolvement**

We came to know that when we scrap our data our website load page from fist page and scrap all data again and again each time loop increments. As I have shown below

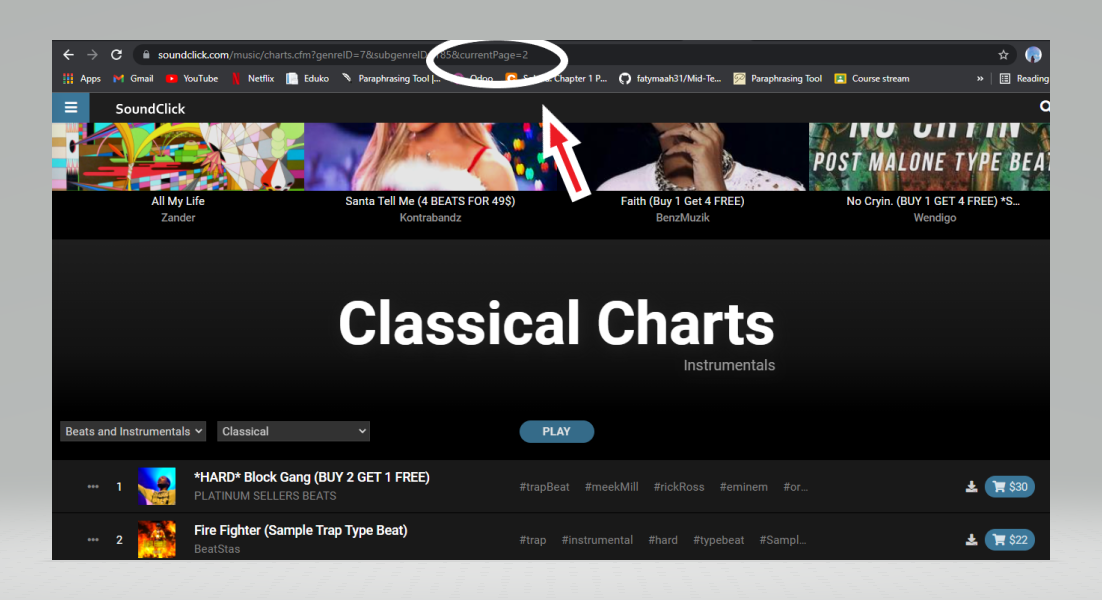


Fig 4‑4 PageNo shown

So after lot of efforts we detected the page number on which our website was not scraping previous data and then incremented loop according to it.

## CSV File

After scraping, we need to save our data so we use CSV file to store our data safely. During execution we save it in this file. Here we are providing reference picture.

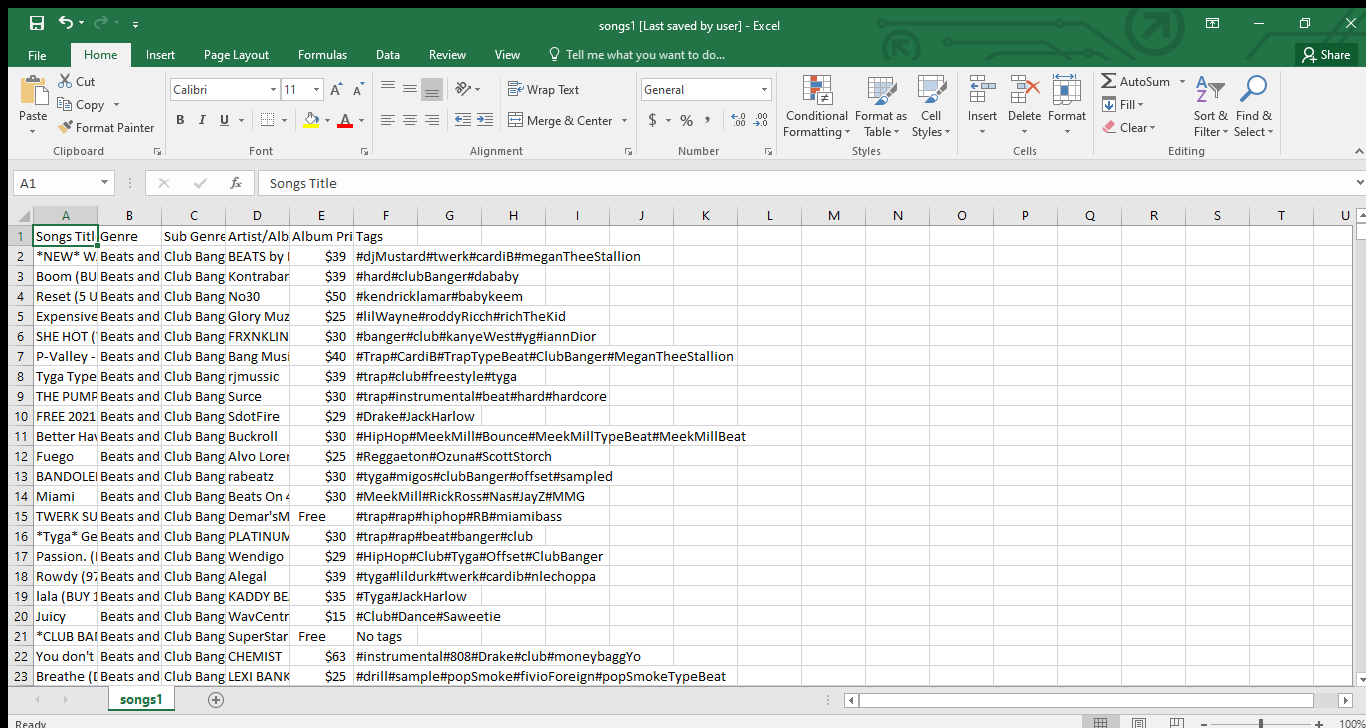


Fig ‑:Reference picture of CSV

## UI Implementation

Building interfaces with an emphasis on aesthetics and interaction is what user interface (UI) design is all about. The objective of the UI designer is to build an interface that is both simple to use and pleasant to the eye.

The access point where a user interacts with a software programme (e.g., Figma, Sketch), a browser-based website, is referred to as the "interface" such as screen of our smart phones.A user interface designer investigates all of a user's interactions and behaviours with a product in order to produce an interface that best suits the user's requirements.

The aesthetic decisions a designer takes while producing a product, such as an image, button, label, are referred to as UI design. All of these factors will have an impact on the user's engagement, thus they must be considered carefully.

### Qt Designer

Qt Designer is a Qt tool for creating and constructing Qt Widget-based graphical user interfaces (GUIs). You may use a what-you-see-is-what-you-get (WYSIWYG) approach to create and personalise your windows or dialogues, and test them in a variety of styles and resolutions.

Using Qt's signals and slots technology, widgets and forms produced with Qt Designer interact smoothly with programmed code, allowing you to simply assign behaviour to graphical objects. All Qt Designer properties may be updated dynamically in the code.

### Pencil Tool

At first place we design our UI in pencil tool which was consider as a rough idea of our Graphical user interface. Here we are providing picture of it.

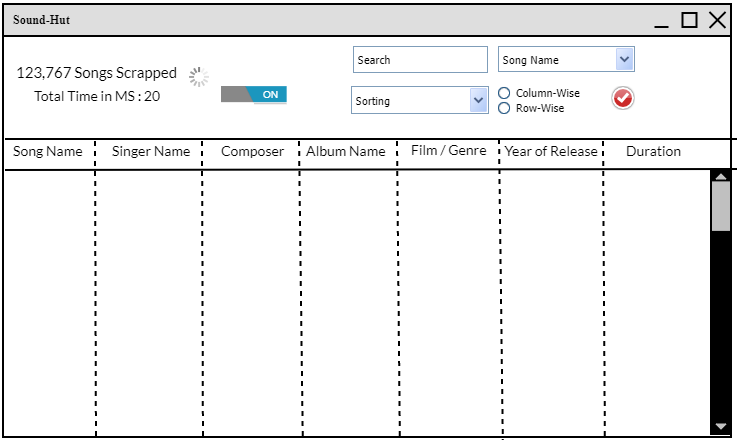


Fig 4‑6:Pencil Tool UI

After Pencil tool we make our GUI in Qt designer and our final GUI resulted as:

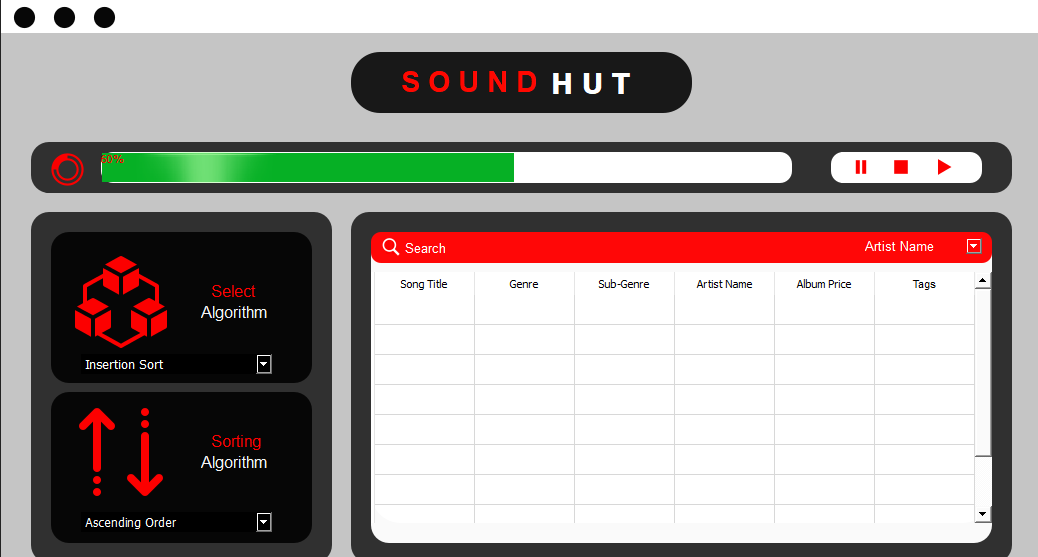


Fig 4‑7: Final GUI

As we can see Pencil tool UI and our Final UI has huge changes we design our Final Ui user friendly so everyone can easily understand it and has access to it. To make our GUI more interactive we use icons and colors in it. To use icons we need QRC file

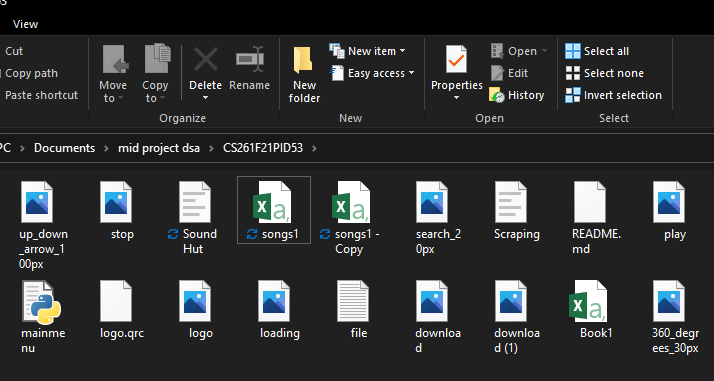


Fig 4‑8: QRC Files

Some of the widgets we used are shown below:

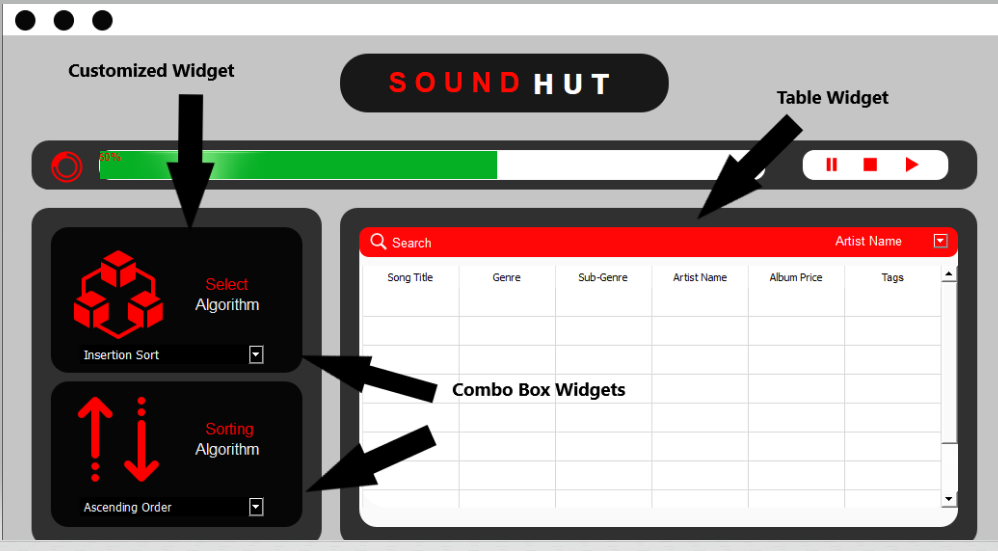


Fig 4‑9: Widget ScreenShot

## Collaboration

We, 2020-CS-138 and 2020-CS-112 collaborated in a professional way. All the implementations we did were discussed together before we implemented them. We divided work equally between us and collaborated in a good manner. Before pencil tool, we discuss with each other that how we want to make our GUI and how it can be convenient for User. Coding was also done in a professional manner. We solved each other issues nicely and consider each other problem our own problem in order for completion of this project.

# Code Execution

After designing UI we move toward Coding. Its brief description is given below

## Converting to Python

First of all we convert our UI into python code. We use anaconda cmd to convert our GUI into python. We use commands to covert this.

## QTable Widget

In order to show our data in table we use Qtablewidget which will help us in showing data. Here we can add number of columns and rows according to our need. Screenshot of scrap data in widget is also given below. Here, Data will be loaded into the table from csv file

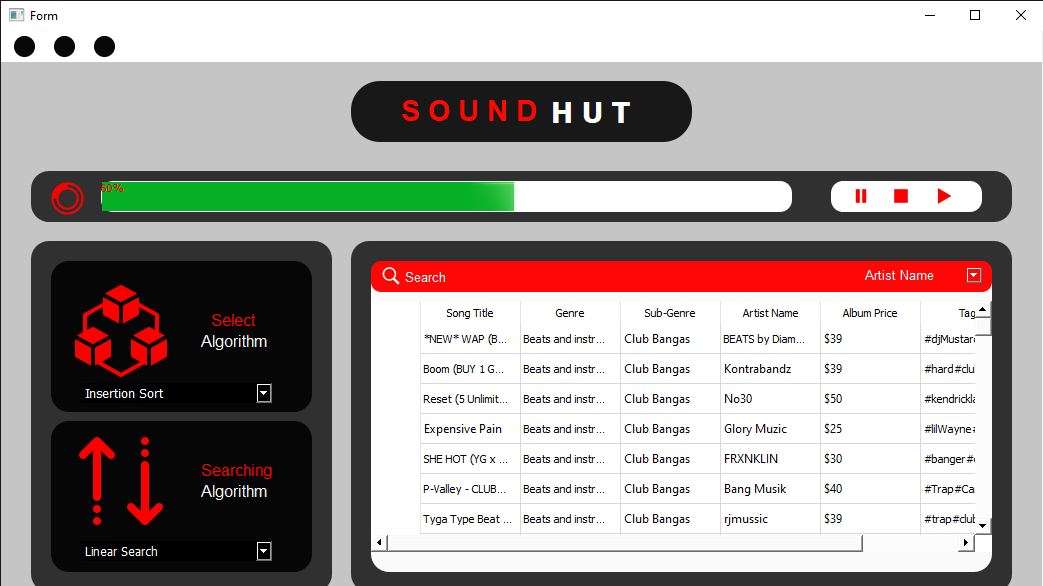


Fig 5‑1: Scrapped Data in Table

## Mouse Events

In GUI we will use mouse event in order to perform our operations. User can click on these icons in order to perform their relative function

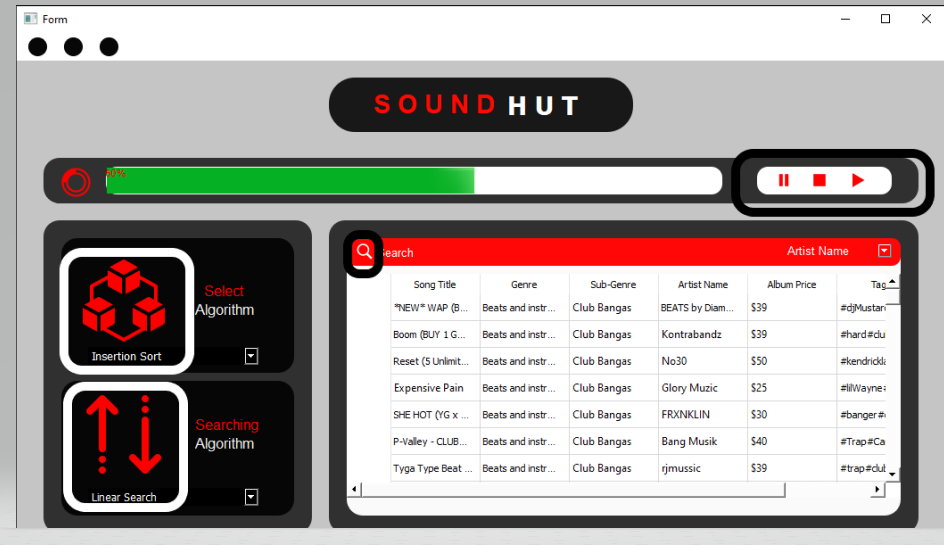


Fig 5‑2: Mouse Event Screenshot

Following operations can be perform using Mouse Events:-

* Start
* Pause
* Stop
* Selecting Sorting Algorithm
* Selecting Searching Algorithm
* Search
* Sort

## Incomplete ideas:

### Progress Bar:

We have to show that how much data is scrapped in progress bar. But due to its complexity and shortage of time we did not complete it. In this we are also looking forward towards pausing and stopping data but we only did completion of start scraping. Screenshot of it is shown below that how we decided to implement it.

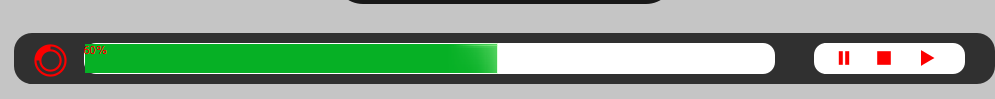


Fig 5‑3: Progress Bar Screenshot

### Multi-Level Sorting:

According to our project requirement we have to done multi-level sorting that if anything is same in one column we have to sort it on the basis of another column but unfortunately we cannot do this due to its complexity.

# User Manual

## Start Scrapping

In order to start scrapping, you have to click on start button as shown in picture. As soon as scraping end data will be loaded in table. Screenshot of it is shown below

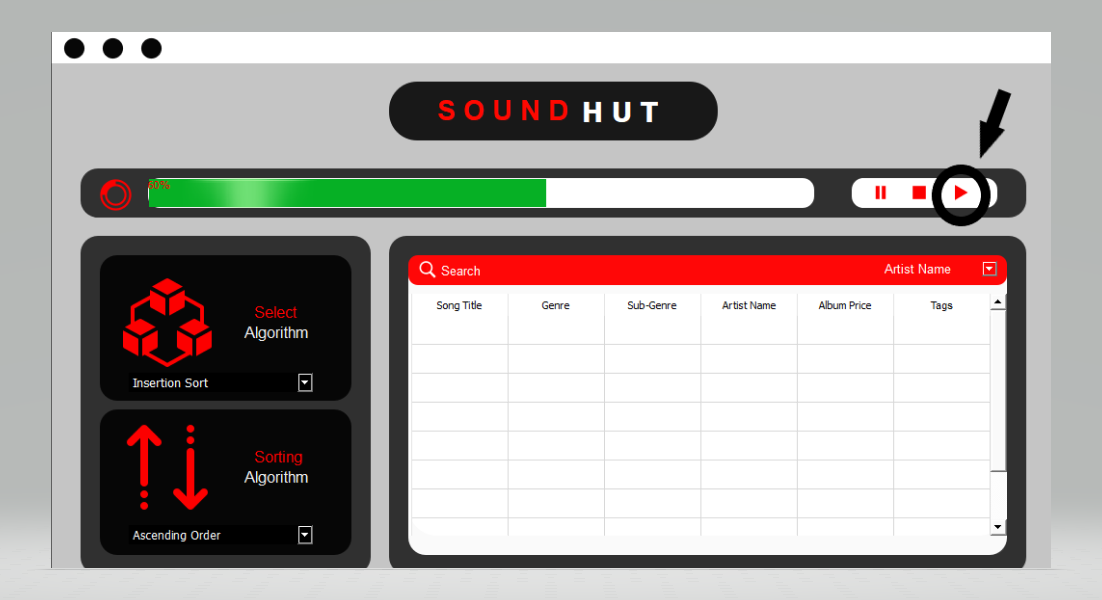


Fig 6‑1: Start Button

As soon as scraping start data will be loaded in table:

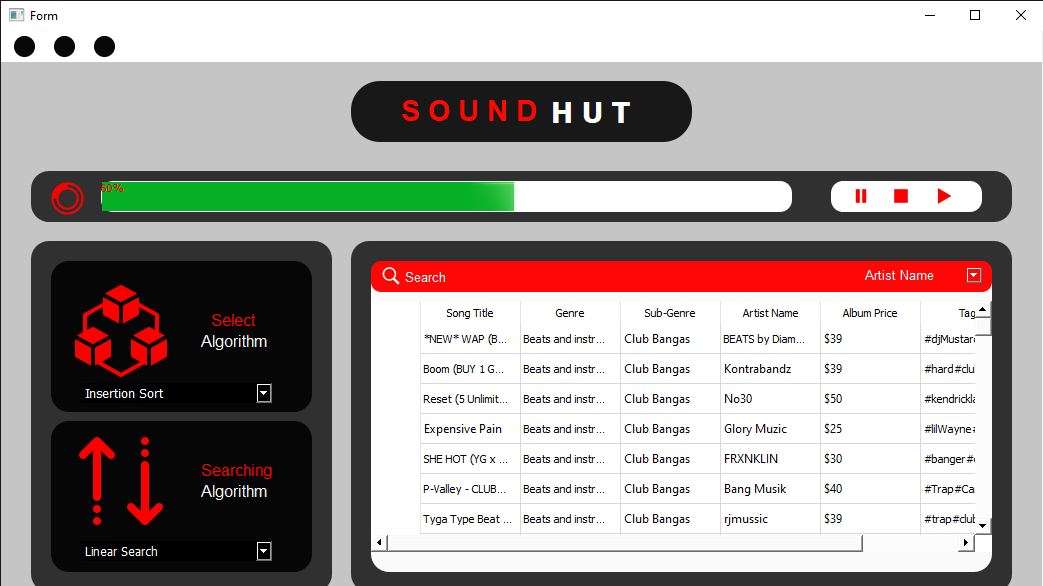


Fig 6‑2: Scrapped Data

## Sorting a Data

To sort data we have to choose algorithm first through combo box. And we can sort it by clicking on its icon

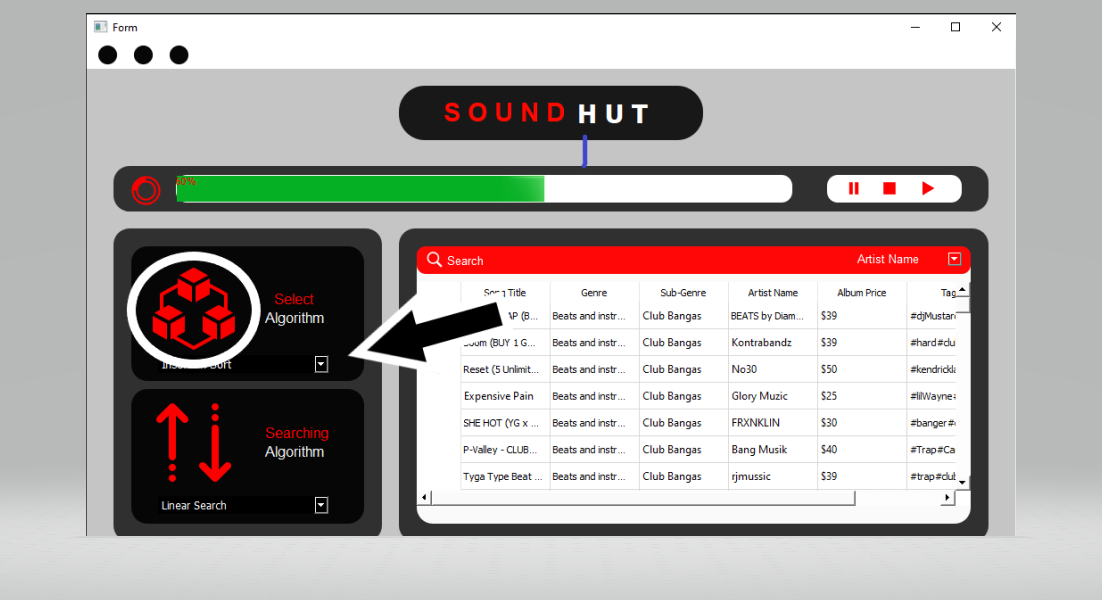


Fig 6‑3: Sorting a Column

## Searching a Data

In order to search data, First we will choose its algorithm and then we will write a word we want to search and choose the column in which we want to search and will search by clicking on search icon

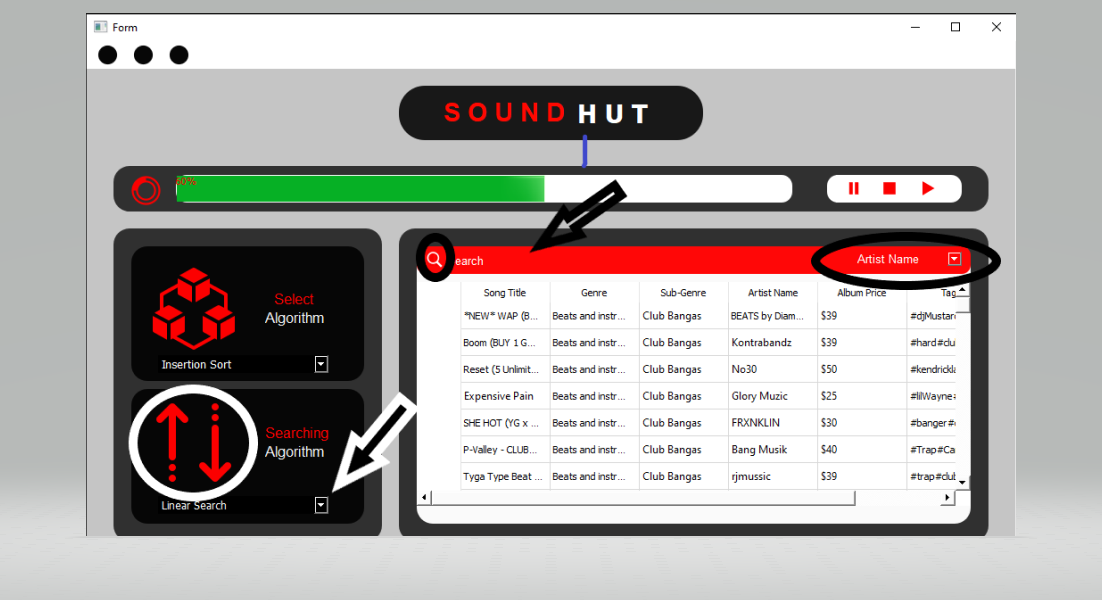


Fig 6‑4: Searching a Data

## Viewing a Data

In order to view all data we need to use scroll bar in table . They are also shown below in screenshot

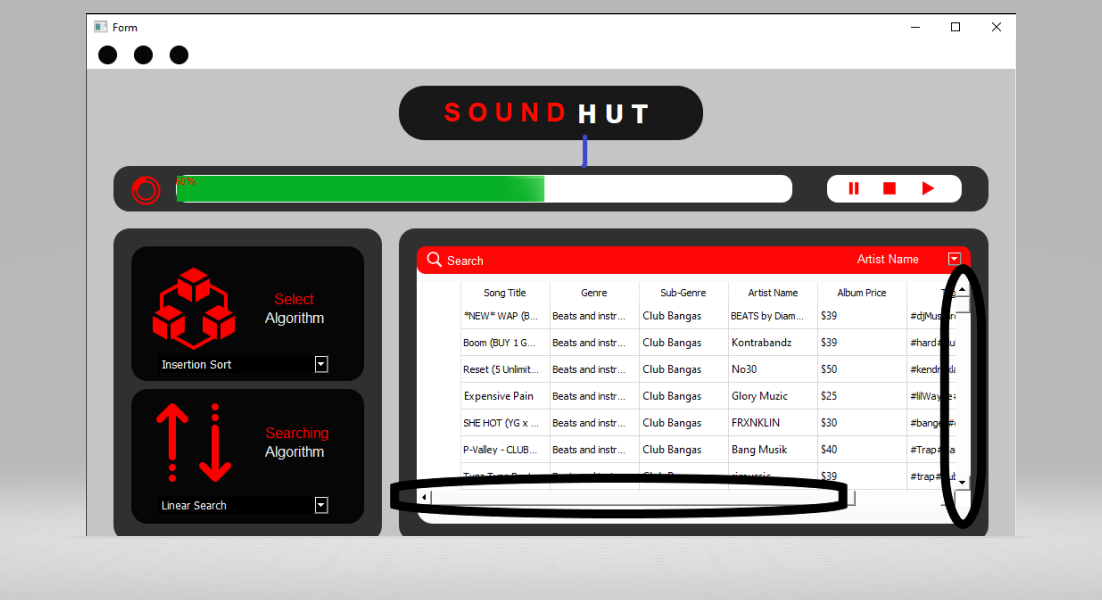


Fig 6‑5: Scrolling Bar

# Task Division

|  |  |
| --- | --- |
| Group Member | Task |
| 2020-CS-138 | Proposal Details |
| 2020-CS-138 | Business Case |
| 2020-CS-112 | Technical Details |
| 2020-CS-138 | Interface for Project |
| 2020-CS-138 & 2020-CS-112 | Scraping coding |
| 2020-CS-138 | Loading Data in CSV |
| 2020-CS-112 | Idea on Pencil Tool |
| 2020-CS-138 | Improving the Idea |
| 2020-CS-112 | GUI Making |
| 2020-CS-112 | Table Widget |
| 2020-CS-138 | Icons and Qrc file |
| 2020-CS-138 | Updated GUI and changes |
| 2020-CS-112 | Description of Half Algorithms |
| 2020-CS-138 | Description of Half Algorithms |
| 2020-CS-112 | Converting GUI into Python |
| 2020-CS-138 | Loading Data into Table |
| 2020-CS-112 & 2020-CS-138 | Coding of Sorting Data |
| 2020-CS-112 | Searching Algorithms |
| 2020-CS-112 | Previous Report analysis and updating |
| 2020-CS-138 | Formatting |
| 2020-CS-138 | Achievement & Integration |
| 2020-CS-138 | Final Recheck |

# Project Performance

|  |  |  |  |
| --- | --- | --- | --- |
| **Challenges** | **Success Criteria** | **Criteria Met** | **Comments** |
| Project Idea | Products must not be general | Yes | Rejected once but second idea was accepted |
| Proposal Report | All details must be given | Yes | Successfully did it. |
| Scraping | One Million data with 7 entities | Maybe | Cannot scrap one million data due to internet issue and slow scraping |
| Pencil Tool | Must be detailed | Yes | Two screens were made but later removed first one |
| GUI | User Friendly, Easy to Access | Yes | Icons and good color scheme was used in order to make it more user-friendly |
| Sorting | 12 sorting algorithms should be used | Maybe | Only few sorting algorithms were implemented |
| Searching | Multiple searching algorithms should be used | No | Not implemented yet |
| Loading Data | Load Data accurately in table | Yes | All data is accurately loaded in table |
| Progress Bar | Can start, pause and stop scraping with time analysis in ms. | No | Not Implemented yet |