Scrap the Movies



Project Supervisor

Mr. Samyan Qayyum Wahla

Project Report

Proposer Details

Group Number (**G19**)

|  |  |
| --- | --- |
| Sharjeel Iftikhar | 2020-CS-139 |
| Saqlain Nawaz | 2020-CS-135 |

Department of Computer Science

University of Engineering and Technology, Lahore Pakistan

**Report Overview**

**Executive Summary**

Program will scrap data from different movies websites such as **YTS**, **MoviesFlix**, and **Melomovies** etc. Scrapped data will include entities such as movie title, movie cast, genre, rating, releasing year, releasing date, movie director and synopsis. We provide a fully user friendly interface in such a way that Program will contain the progress bar of scrapped movies and obviously it will have a facility to pause the scrapping and perform the other features such as sorting and searching and simply start when he wants.

User can **sort** the entities by applying algorithm of his own choice such as by

* Merge sort
* Insertion sort and much more.

And on time of search user also have the facility to apply filters on his search such if he is searching the movie by its title then we’ll provide him facility to search like that if he writes character **A** then program will show him all the movie titles starting with, contain with and end with character **A.**

User also sort the entities according to **Ascending or Descending** order on just one click and there will also multi-Colum sorting in the program.

User can also **save the file** whenever he wants. We also provide him a feature of searching by the basis of his own selected entity such as title wise or on basis of genre or on basis of rating. Program also calculate the time taken by the particular task in seconds.

One of the benefit of this program that we are going to provide to user is that we will help the user to see what type of movies are now on trending and which movie’s rating hold good numbers and on which movie’s genre he should invest. And by with this he can also see that which actor is public’s favorite. By this he can cast him in his next project etc.

**Business Needs**

Our project will be focusing on both any movie production company and a random user.

If a Production Company is going to view the details of the movies that remain top on rating in recent years and on average which type of genre will be profitable for them according to statistics. Then our project will provide them all the statistics according to their needs.

User can also use this project to search the movies of his own desires from data of movies around **1 million** according to his preferences.

**Domains in Real Life**

* For user real life domain is **Entertainment.**
* Life domain for film industries is select best choice for their **business**.

**Motivation for Project**

Firstly almost all the people have great interest in movies including myself. And with the **completion** of this project we got almost data of 1 million movies in our pc and so we can find any kind of movie without having the access over **Internet.** Secondly as it is mentioned in project requirement that you have get 1 million scrapped data so Movies was the **second** option came in my mind, first one was different products from shopping websites. But our first option we already done practice of it in lab task .And finally it fits according to our interest.

**A film is the petrified fountain of Thought.**

**Jean Cocteau ….**

**Audience**

Actors for project will be

* Streamer
* Film Industry
* Director

**Streamer** willbe any random movie fan who willing to find the best movies of his own taste by applying different filter provide in the project.

**Film Industry** in such a way that if particular film industry hiring a best movie director for their movie, then this project will meet all their needs. As similarly the **Director** will select the cast that perform outclass in previous films.

**Technical Details**

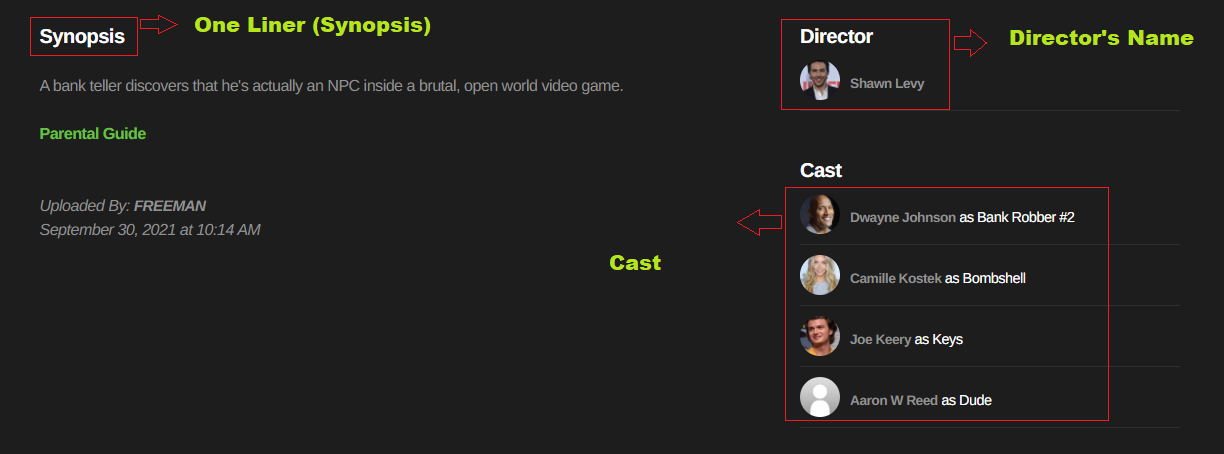
**Entities**

|  |  |  |
| --- | --- | --- |
| ***Name*** | ***Data Type*** | ***Description*** |
| Title | String | It will be Name of the movie |
| Duration | String | Move timing in minutes. |
| Year | String | In string because some movies have year duration such as (2010-2015) |
| Genre | String | Topic of life movie covered. |
| Director | String | The movie creator director. |
| Cast | String | Leading actors and actress |
| Synop | String | One liner of the movie. |

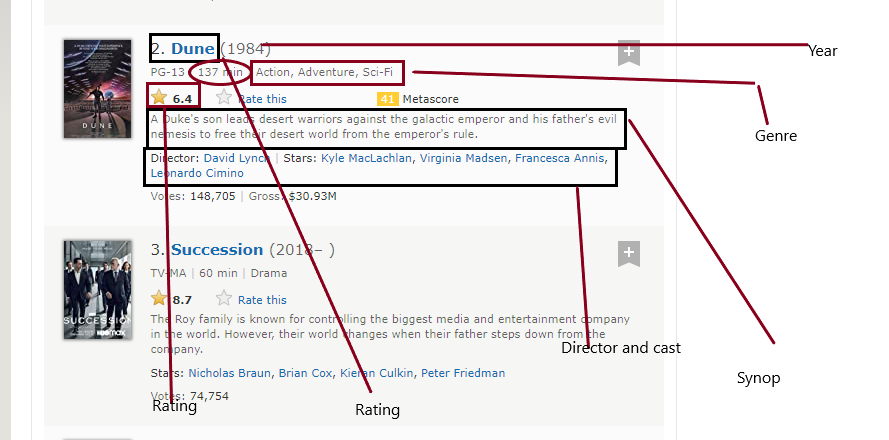
**Sample of Scrapping Source**

First I started scrapping form https://yts.mx/ and there is the position of the required entities that I scrapped.

**



But the drawback of this website is it contain only 35k movie and also most of the movie details director, cast as well as synopsis was missing. That’s why I shifted to Imbd website which is all time best website about movies.



Here we find around more than 6 million movie by apply only one filter.

**Sorting Algorithms**

**Insertion Sort**

|  |  |
| --- | --- |
| **Description** | The basic idea of this algorithm is to divide into an **unsorted** and sorted region. Initially, the first term is the only member of the sorted region. The insertion sorts repeatedly scans the list of items, each time inserting the item in the unordered sequence into its correct position.  Then it find the correct place for a given element in a sorted list. |
| **Pseudo Code** | Insertion\_Sort(A):  for i=1 to A.length  key=A[i]  //Insert A[i] into sorted sequence A[1….j-1]  j= i–1  while j>=0 and A[j]>key  A[j+1] =A[j]  J=j–1  A[j+1]=key |
| **Code** | def Insertion\_Sort(arr):      key=0      for j in range(1,len(arr)):          key = arr[j]          i=j-1          while((i>0 or i==0) and arr[i]>key):              arr[i+1]=arr[i]              i=i-1          arr[i+1]=key      return arr |
| **Time Complexity** | The **best-case** time complexity of Insertion Sort is: ***O(n).***  The **worst-case** time complexity of Insertion Sort is: ***O(n²).*** |
| **Proof of Correctness** |  |
| **Three Strengths** | * Space requirement is minimum. * **Stable** doesnotchange the relative order of key. * Main advantage of insertion sort is its simplicity and efficiency. * It has good ability to sort small lists. |
| **Three Weakness** | * It requires **n2** number of steps for sorting. * It does not perform well with large numbers. * It is not friendly for **practical** applications. |
| **Dry Run** |  |

**Merge Sort:**

|  |  |
| --- | --- |
| **Description** | This algorithm is based on **divide and conquer** rule. **Firstly,** we **s**tartdividing the array into two equal parts until we get the single element in divided arrays. This is called dividing part. Then comes the **Conquer** partinwhich we merge the divided parts of the array into final sorted array. |
| **Pseudo Code** | **MERGE-SORT(A, p, r):**  If p > r  return  q = ( p + r) / 2  MERGE\_SORT(A, p , q )  MERGE\_SORT(A, p +1 , q )  Merge(A, p, q, r )    **Merge( A, p , q , r )**  n1 = q – p + 1  n2 = r – q  Let L[ 1…n1 + 1] and R [1… n2 +1 ] be new arrays  ***For*** *i = 1* ***to*** *n1*  *L[i] = A[ p + i -1 ]*  ***For*** *j = 1* ***to*** *n2*  *R[j] = A[ q + j]*  *i = 1*  *j = 1*  *for k = p* ***to*** *r*  ***if***  *L[i] ≤ R[j]*  *A[k] = L[i]*  *i = i + 1*  ***else*** *A[k] = R[j]*  *j = j + 1* |
| **Code** | #Merge SOrt arry  def Merge\_Sort(arr,start,end):      if(start<end):          mid = (start+end)//2          Merge\_Sort(arr,start,mid)          Merge\_Sort(arr,mid+1,end)          Merge(arr,start,mid,end)      else:          return  function merge  def Merge(arr,start,mid,end):      start -=1      mid -=1      end -=1      left\_arr=[]      right\_arr=[]      for i in range(start,mid+1):          left\_arr.append(arr[i])      for j in range(mid+1,end+1):          right\_arr.append(arr[j])      i=0      j=0      k=start      for k in range(start,end):          while i < len(left\_arr) and j < len(right\_arr):              if(left\_arr[i]<right\_arr[j] or left\_arr[i]==right\_arr[j]):                  arr[k]=left\_arr[i]                  i = i+1              else:                  arr[k]=right\_arr[j]                  j=j+1              k=k+1  # Checking if any element was left          while i < len(left\_arr):              arr[k] = left\_arr[i]              i += 1              k += 1          while j < len(right\_arr):              arr[k] = right\_arr[j]              j += 1              k += 1      return |
| **Time Complexity** | Time complexity of Merge Sort is **O (n log n)** in **best and worst** both case. |
| **Proof of Correctness** |  |
| **Three Strengths** | * It has better run time than insertion, selection and bubble so it is quicker in implementation * It is also efficient for large numbers **n,** andalsois **stable.** * It has constant run time. |
| **Three Weakness** | * Even if we provide sorted array it run through whole process. * Slower comparative to the other sort algorithms for smaller tasks. * Use more memory space to store initial split arrays during **dividing step.** |
| **Dry Run** |  |

**Selection Sort:**

|  |  |
| --- | --- |
| **Description** | The selection sort works by repeatedly going through the list of items. In the first step we find the **minimum** element from the array and put it in the declared variable in which store the minimum value we find, and at the last of the array the digit stored in declared variable of min is placed at the **beginning** of the array. Now the iteration occur and now the complete above process repeated for the next digit and so on for the last element of the array.  . |
| **Pseudo Code** | *Selection\_Sort(Arr):*  *n = size of Arr*  *for i = 0 to n-1*  *#declare a variable to store the min value*  *min = i*  *# another for loop to check whole array*  *for j = i+1 to n-1*  *if Arr[j] > Arr[i]*  *min = j*  *# Now arrange the position of min value by swapping*  *if min != i*  *swap Arr[min] and Arr[j]* |
| **Code** | # Selection Sort  def Selection\_Sort(arr):      n=len(arr)      for i in range(n):          min = i          for j in range(i+1,n):              if (arr[min]>arr[j]):                  min=j           # Swapping the elemnts          arr[i],arr[min] = arr[min],arr[i]      return |
| **Time Complexity** | Worst case of Selection sort is **O(n2).**  Best case of Selection sort is also **O (n2).** |
| **Proof of Correctness** |  |
| **Three Strengths** | * It uses few operations, so where data movement is costly it is more useful. * It does not depend upon the **initial arrangement** of the data. * It is better for small no of arrays. * Does not need any extra memory |
| **Three Weakness** | * It is not **stable.** * Not suitable for large numbers because of **n2** runtime. * Even if we provide sorted array it run through whole process. |
| **Dry Run** |  |

**Bubble Sort:**

|  |  |
| --- | --- |
| **Description** | Bubble Sort is also referred as **Sink\_Sort.** It is very simple sorting algorithm. It is quite familiar with the insertion sort, the main difference is that in insertion we compare the key to the whole elements of array while in bubble sort we only compare the **adjacent pairs** and swap it according to particular condition. This passing procedure is repeated until no swaps are required, indicating that the list is sorted. In this algorithm the small digits are gradually start moving top of the list. |
| **Pseudo Code** | Bubble\_Sort(arr)  c = arr .length  for i to c -1  flag = false  for j = 0 to c-1  if arr[ j ] > arr[ j+1 ]  swap arr [ j ] and arr [j + 1 ]  flag = true  if flag == false  break  return arr |
| **Code** | # Bubble Sort  def Bubble\_Sort(arr):      control=len(arr)      for i in range(control-1):          flag=False          for j in range(control-1):              if(arr[j]==0):                  temp=arr[j+1]                  arr[j+1]=arr[j]                  arr[j]=temp                  flag=True          if(flag==False):              break      return arr |
| **Time Complexity** | The **best-case time** complexity of Bubble Sort is: **O (n).**  The **worst-case time** complexity of Bubble Sort is: **O (n²).** |
| **Proof of Correctness** |  |
| **Three Strengths** | * It is simple to **implement** and easy to understand. * Best advantage of Bubble sort is memory space it require very few space and does **not** need any **temporary memory**. * When we provide sorted array its run time is **O (n)** which is its best case. |
| **Three Weakness** | * Its worst case is **O (n2).** * It is not very useful in practical application (real world). * It does not deal well with a list containing a huge number of items.   . |
| **Dry Run** |  |

**Quick Sort:**

|  |  |
| --- | --- |
| **Description** | Quick Sort algorithm is also based on **divide and conquer** rule. In this algorithm we divide the array into two parts just like we did in merge sort but the difference here is we divide on the basis of the **pivot.** The pivot element could be last element of the array or the middle element or the first element of the array. But here I am using the last element as a pivot.  Then according to the pivot element we arrange the array such that elements less than pivot came on **left** and large digits came on right side of the pivot. Then we recursively call the function until our starting index **(low)** < to the end index **(high)**. When this condition fails we find our sorted array. |
| **Pseudo Code** | QuickSort(A ,p ,r):  if p<r:  q=Partition (A, p, r)  QuickSort(A, p,q-1):  QuickSort(A, q+1, r):  Partition (arr , low, high)  // pivot (Element to be placed at right position)  pivot = arr[high];  i = (low - 1) // Index of smaller element and indicates the  // right position of pivot found so far  for (j = low; j <= high- 1; j++)  // If current element is smaller than the pivot  if (arr[j] < pivot)  {  i++; // increment index of smaller element  swap arr[i] and arr[j]  swap arr[i + 1] and arr[high])  return (i + 1) |
| **Code** | #Quick Sort  def quick\_Sort(arr,low,high):      if(low<high):          pi=partition(arr,low,high)          quick\_Sort(arr,low,pi-1)          quick\_Sort(arr,pi+1,high)  Partition Function  def partition(arr,low,high):      # pi is partitioning index, arr[pi] is now at right place      pivot=arr[high]      i=low-1      for j in range(low,high):      # If current element is smaller than the pivot        if(arr[j]<pivot):              i+=1              temp=arr[i]              arr[i]=arr[j]              arr[j]=temp       # Smaller part end and now we swap pivot to its correct place      temp=arr[i+1]      arr[i+1]=arr[high]      arr[high]=temp      return i+1 |
| **Time Complexity** | Time complexity of Quick Sort in **best-case** is **O (n\*logn)**.  Time complexity of Quick Sort in **worst-case** is **O (n²)**. |
| **Proof of Correctness** |  |
| **Three Strengths** | * It is said to be the best sorting algorithm. * It is able to deal well with huge lists because its worst case is also **n\*log n**. * Space friendly, No additional storage is required. |
| **Three Weakness** | * Its worst-case performance is equal to average case performance of insertion sort. * It is **not stable** rather it is **fragile**. * If the list is already sorted than bubble sort it is not much more **efficient**. * It also not deal well with **negative** integers. |
| **Dry Run** |  |

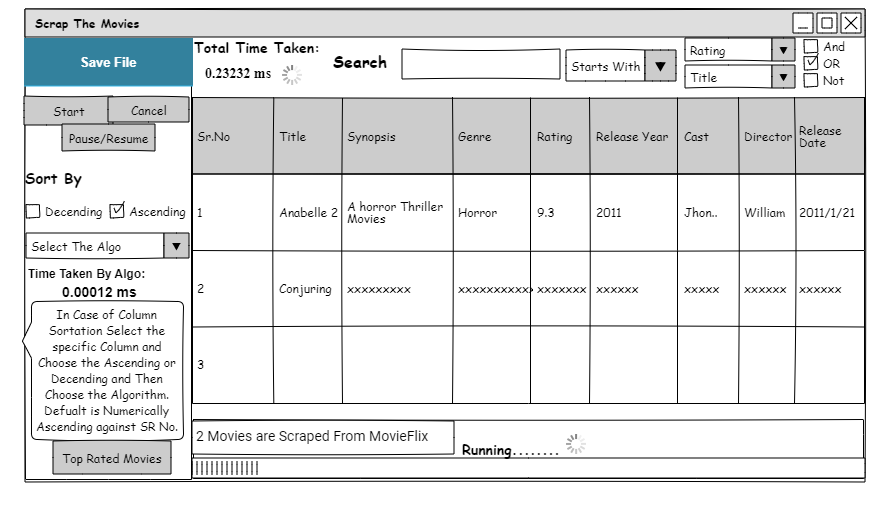
**Hybrid Sort:**

|  |  |
| --- | --- |
| **Description** | This is the combination of Quick Sort and Insertion sort. As we know Insertion sort perform very well on partially sorted arrays and quick also is also perform nlog(n) times when partition is balanced so we make a hybrid of both of them. It perform in this manner that if our first array length is > 9 then we perform first the quicksort then we check if left one or right one is smaller and on smaller array we perform insertion sort and so on. |
| **Pseudo Code** | QuickSort(A ,p ,r):  if p<r:  if (low +high +1<10)  insertionSort(arr,low,high)  else  if (pivot –low < high – pivot )  hybrid\_Sort(arr,low,pivot-1)  else  opposite of above  Partition (arr , low, high)  // pivot (Element to be placed at right position)  pivot = arr[high];  i = (low - 1) // Index of smaller element and indicates the  // right position of pivot found so far  for (j = low; j <= high- 1; j++)  // If current element is smaller than the pivot  if (arr[j] < pivot)  {  i++; // increment index of smaller element  swap arr[i] and arr[j]  swap arr[i + 1] and arr[high])  return (i + 1) |
| **Code** | def Insertion\_Hybrid(self,arr,low,high,col):          #key=0          for j in range(low+1,high+1):              key = arr[j]              i=j-1              while((i>j or i==j) and arr[i][col]>key[col]):                  arr[i+1]=arr[i]                  i=i-1              arr[i+1]=key          return      # Hybrid Quick Sort (Quick and Insertion)      def Asc(self,arr,low,high,col):          print(high)          while(low<high):              # First check if the array is smallr enough(less than 10) to apply insertion sort or not              if(high-low+1<10):                  self.Insertion\_Hybrid(arr,low,high,col)                  break              else:                  pi=self.partition\_Asc(arr,low,high,col)                  # NOw find out on both sids of pivot that which array is larger so that we can apply                  # recursive hybridsort on it                  if(pi-low<high-pi):                      self.Asc(arr,low,pi-1,col)                      low = pi +1                  else:                      self.Asc(arr,pi+1,high,col)                      high=pi-1 |
| **Time Complexity** | Time complexity is **O(n2).** |
| **Proof of Correctness** |  |
| **Three Strengths** | * It is able to deal well with huge lists because its worst case is also **n\*log n**. * Space friendly, No additional storage is required. * Main advantage of insertion sort is its simplicity and efficiency. |
| **Three Weakness** | * It is not friendly for **practical** applications. * It requires **n2** number of steps for sorting. |
| **Dry Run** |  |

**Tree Sort:**

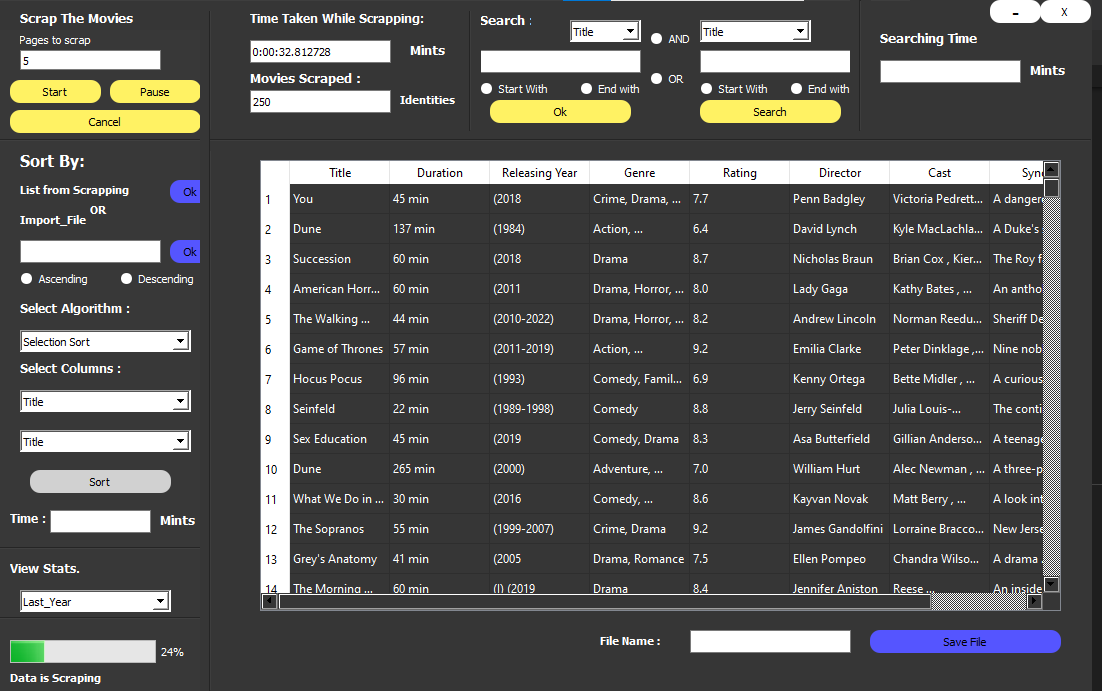
|  |  |
| --- | --- |
| **Description** | Tree sort is an online sorting algorithm. It uses the binary search tree data structure to store the elements. The elements can be retrieved in sorted order by doing an in-order traversal of the binary search tree. Since it is an online sorting algorithm, the elements inserted are always maintained in sorted order. |
| **Pseudo Code** | Class Node:  Data  Node.left  Node.right  Class Tree:  Root=null  Coming node is smaller or equal to root then add it to left child  Else wise add to right side  Inorder traversal for ascending |
| **Code** | class Tree:      def \_\_init\_\_(self):          self.root=None      def insert\_node(self,ddata,root,col):          if(self.root==None):              self.root=ddata          else:              if(ddata.data[col] < root.data[col] or ddata.data[col] == root.data[col]):                  if(root.left==None):                      root.left=ddata                      ddata.parent=root                  else:                      self.insert\_node(ddata,root.left,col)              elif(ddata.data[col] > root.data[col]):                  if(root.right==None):                      root.right=ddata                      ddata.parent=root                  else:                      self.insert\_node(ddata,root.right,col)      def in\_order\_traversal(self,node,col,arr):          if(self.root==None):              print("tree is empty")          else:              if(node.left!=None):                  self.in\_order\_traversal(node.left,col,arr)              arr.append(node.data)              #print(node.data)              #print()              if(node.right!=None):                  self.in\_order\_traversal(node.right,col,arr)          return arr |
| **Time Complexity** | Worst case is **O(n2)**  Best case is **n(logn)** |
| **Proof of Correctness** |  |
| **Three Strengths** | * Tree sort algorithm is as fast as quick sort algorithm. * Tree sort is stable * Best for large number of arrys |
| **Three Weakness** | * Worst case occur when the elements in an array is already sorted. * In the running time of tree sort algorithm is O (n2) |
| **Dry Run** |  |

**UI**

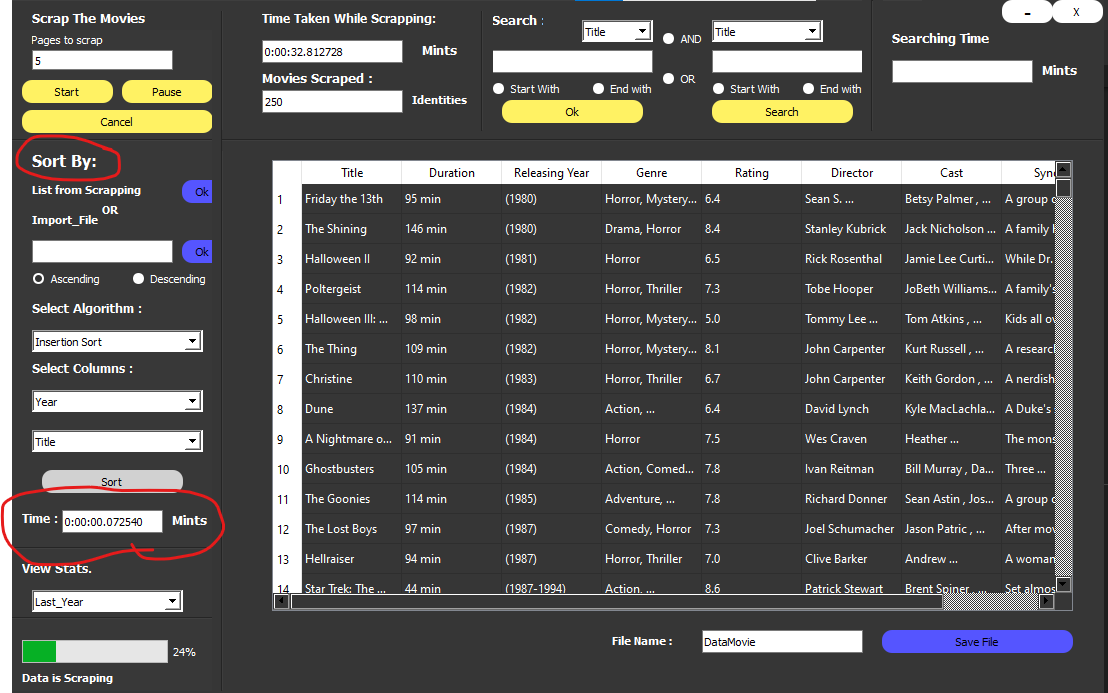
****Wireframes**

**Implemented UI:**

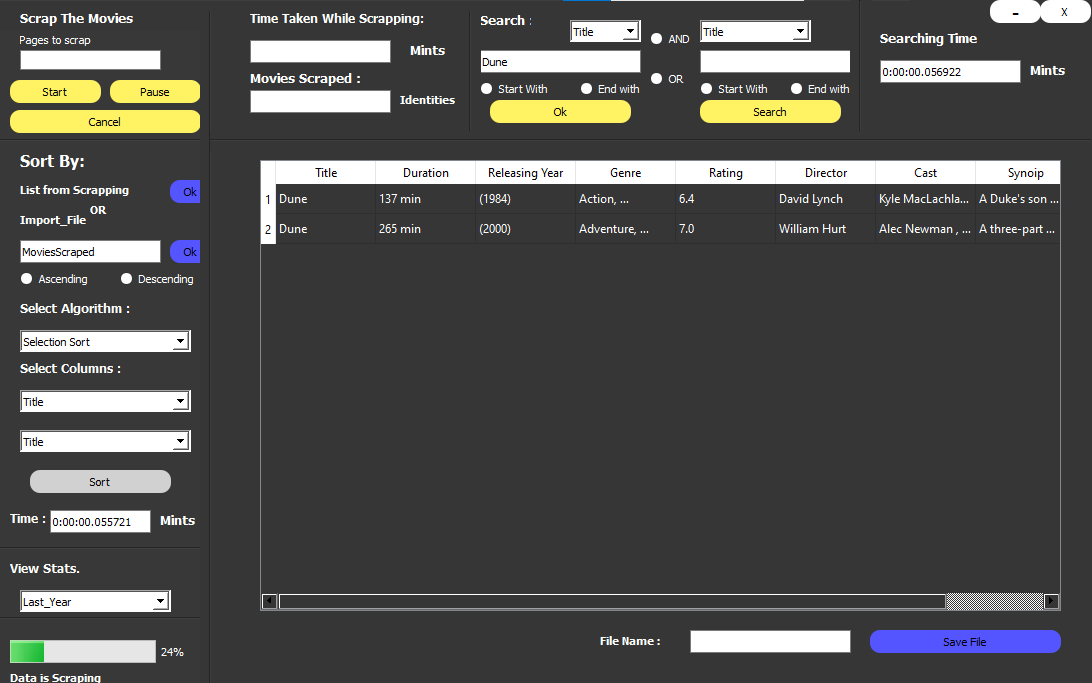
Scrapping Window

****

Sorting window

****

Searching Window

****