

Movies Scrapper

Project Report

Project Supervisor

Mr. Samyan Qayyum Wahla

Project Members(**G-07**)

Umair Manzoor 2020-CS-24

Department of Computer Science

University of Engineering and Technology, Lahore Pakistan

# Project Overview:

## **Executive Summary**

IMDB is a vast database of movies, dramas and TV shows of all types. Now-a-days such things are a primary source of entertainment for almost everyone. Therefore, if someone is looking for movies or searching for genre or actor he/she may seek help of this software.

This software scrapes data which is the titles of movies along with additional information. These entities are genre, actors, releasing dates, Ratings, synopsis and much more which would help to sort the immense data to specifically look for the movies that the user desires and generates the scraped data of the movies. I’ve implemented a GUI that would compromise of functions such as to commence the process of the scraping, pause and halt the system whenever the user desires.

Users then can sort this data according to their priorities. It’ll offer these features for sorting.

* 8 Algorithms
* Ascending & Descending Order

User can trace time consumed & progress during this process.

Similarly, User can search this huge data and filter his search accordingly. Search provides following features:

* Search Single column
* Search using starting and ending keyword.

Using these features one can use this software according to his need like finding movies or shows to watch, looking for top rated movies, tracing directors and finding movies of his favourite actor, search movies released in particular year.

## **Motivation:**

Multi-media is fashion of the day. Whenever we hear word entertainment, movies and TV shows pop up in our minds. Market of Streaming platform and app is very trending these days. Millions of movies are out there. People actually love to spend time on such sort of entertainment. So, I thought it would be beneficial if there’s tool which can provide information on such things effectively and accurately. Thus my thought was fixated on movie scraping and extracting information out of it. Also, I myself like movies and often face difficulty to find “what-to-watch”.

*“A good movie can take you out of your dull funk and the hopelessness that so often goes with slipping into a theatre.”*

## **Objectives:**

* Full Scale Scraping (That will scrap all necessary information)
* Efficient Sorting Algorithms
* Searching
* User Friendly Interface

## **Audience:**

End users can be

* Streamers
* Directors
* Writers
* Laypeople

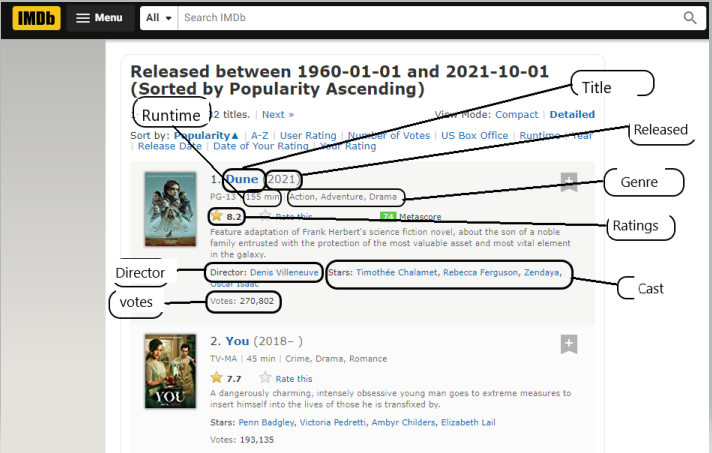
Different users can use this according to their preferences. As streamers and movie sites can use this as a tool to find new trends, follow filmography of celebrities and directors. Writers can carve different stories to set new trends by following current one. Similarly, laypeople can use to find “What-to-Watch tonight”.

## **Business Need:**

A movie streaming platform would be a great business in the need of this type of software. This system would match the preference of the users of the platform to list the desired movies. It’ll provide all necessary information efficiently and provides business with data to determine statistical analysis of productions.

Also Movie Industry can be followed thoroughly just if some modifications are made and it’ll help out industry also in this regard.

**Scraping Site:**



Source: <https://www.imdb.com>

Figure 1: IMDB (site) used for scraping and entities that are scraped. Their detail is given below.

|  |  |  |
| --- | --- | --- |
| **Name** | **Data Type** | **Description** |
| Title | String | Contains Movie name |
| Year | Int | It would consist of the released date of the movie |
| Genre | String | The type and category of the movie. |
| Cast | String | The casting in the movie. |
| Directors | String | The people or person that directed the movie. |
| Synopsis | String | The synopsis (overview) of movie. |
| Rating | float | Ratings according to IMDB in range 1-10 |

## **Algorithms**

|  |  |
| --- | --- |
| **Selection Sort** |  |
| **Description** | This algorithm works by finding minimum element in the unsorted part of array and then it puts it in the beginning of the array. Similarly, if there’s another smaller element than current element at the beginning it then moves it to the beginning. In this way it can pick item form unsorted part and compare it with the sorted part and then moves elements to their appropriate positions. It is an “**unstable**” algorithm based on placing and moving of array elements. |
| **Pseudo Code** | Selection-Sort(Array):  n = Array.length()  for i = 0 to n-1:  minmum\_index = i  for j = i+1 to n:  if Array[j] <= Array[min]:  // store index (j) into minimum  If minimum != i:  // swap i with minimum  return Array |
| **Code** | def selection\_sort(A):  n = len(A) # length  for idx in range(n-1):  min = idx  for j in range(idx+1, n):  if A[j] <= A[min]: # smaller element found  min = j # store index  if min != idx: # swap  temp = A[idx]  A[idx] = A[min]  A[min] = temp  return A |
| **Time Complexity** | Best-case time complexity: O(n²)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** | **Initialization:** when we’re at first index i = 0 Sub-array has one element and one element is always sorted.  **Maintenance:** when first iteration begins we move from i = 0 to i = n and at each iteration if A[minimum] is larger than any other element then it is swapped with that element. Thus, the sub-Array is always sorted.  **Termination:** Outer Loop terminates when i = n. Now when i > n (A.length) the previous sub-Array is sorted and the largest value is at the last index thus we don’t have to check the last element as it is moved to its rightful place. Whole array is sorted.  Thus, Algorithm is correct. |
| **Advantages** | * It is best for small array (little number of inputs). * No additional space is required as we’re manipulating a single array. * if array is already sorted it gives linear runtime[O(n)]. |
| **Disadvantages** | * It requires n2 number of steps for sorting array. Which is worse in accordance with other algorithms out there. * It is at its worst when there’s an array with greater number of inputs. * It is less efficient as its performance is dependent on the initial arrangement of array. |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Bubble Sort** |  |
| **Description** | This algorithm work by continuously swapping two elements in an array based on their comparison with each other. It continues until all the elements aren’t in perfect order. Its execution takes almost same time as of Selection-Sort but it is a “**Stable**” algorithm. Since it uses flag for swapping if array is already sorted it takes less time for its execution. |
| **Pseudo Code** | Bubble-Sort(Array)  n = Array.length  for i = 0 to n-1 do:  swapped = false  for j = 0 to n-1 do:  // compare elements  if Array[j] > Array[j+1] then  // swap  swap(Array[j], Array[j+1] )  swapped = true  if(not swapped) then  break  return Array |
| **Code** | def bubble\_sort(A):  n = len(A) # length of Array  for i in range(n-1):  swaped = False # flag  for j in range(n-1):  if A[j] >= A[j+1]:  # swap with next element  temp = A[j+1]  A[j+1] = A[j]  A[j] = temp  swaped = True  # if not swaped then list is in order  if swaped == False :  break  return A |
| **Time Complexity** | Best-case time complexity: O(n)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** | **Initialization:** Initially the subarray A[1..i−1] is empty. This is the largest element of the subarray.  **Maintenance:** After the execution of the inner loop, A[i] will be the smallest element of the subarray A[i..n]. And in the beginning of the outer loop, A[1..i − 1] consists of elements that are smaller than the elements of A[i..n]A[i..n], in sorted order.  **Termination:** The loop terminates when i = A.length. At that point the array A[1..n]A[1..n] will consists of all elements in sorted order. |
| **Advantages** | * When list is small it is best because it then can sort list efficiently. * It is simple to implement and easy to understand. * it costs a very minimum memory overhead. |
| **Disadvantages** | * it is not suitable for real life applications, due to its time complexity. * It takes more operations when input list is larger thus consuming more of the time. * for every array with n inputs it takes n2 operations to sort. |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Insertion** **Sort** |  |
| **Description** | It checks element for its rightful place in list and places it there. This setup continues for each element and at each iteration. Like previous algorithms its **stable.** It has same Runtime as previous algorithms. |
| **Pseudo Code** | Insertion-Sort(Array):  n = Array.length  for j = 2 to n  min = Array[j]  // Insert Array[j] into the sorted sequence  j = i – 1  while i > 0 and Array[i] > min  Array[i+1] = Array[i]  i = i – 1  Array[j+1] = min  return Array |
| **Code** | def insertion\_sort(A):  for i in range(1, len(A)):  key = A[i]  j = i - 1  while j >= 0 and A[j] > key:  A[j+1] = A[j]  j = j - 1  A[j+1] = key  return A |
| **Time Complexity** | Best-case time complexity: O(n)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** | **Initialization**: The subarray consists of first element and it is sorted since it is a single element.  **Maintenance**: In each iteration element gets inserted into the array only when it is greater than the element to its left. Since the elements to its left have already been sorted, it means is greater than all the elements to its left, so the array remains sorted.  **Termination**: When it reaches last element in the array, which means entire array is sorted. As sub-Array has expanded and covered the whole array. |
| **Advantages** | * Efficient for small number of input. * Simple to understand and implement. * Efficient in system where data insertion is minimal. * Doesn’t use any additional memory. |
| **Disadvantages** | * Runtime complexity limits its uses. * Bad for large number of inputs. * Even in average case its complexity is O(n2) |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Merge Sort** |  |
| **Description** | Merge sort is quite efficient algorithm. It uses Divide & Conquer approach. It is Recursive algorithm that Divides array into sub-arrays and then after placing them into right place it merges them again into a single array thus giving the output. It’s a **stable** algorithm. It best and worst case complexity is same which leaves some loose ends. |
| **Pseudo Code** | Merge-Sort(A, left, right)  n = A.length  If n = 1 // Base Case return single element  Return A  Else // size is greater than make more partitions  mid = (left + right)/2 // find middle  Merge-Sort ( A , left , mid ) // make left and right  Merge-Sort( A , mid + 1 , right) //  Merge(A, left , mid , right)  Merge(A,low,m,high):  Right array of size m+1-low  Left array of size high-m  for i = 0 to low to m+1:  L[i] = A[i]  for j= m+1 to high+1  R[i] = A[j]  i=0  j=0  k = low  while i < L.length:  A[k] = Left[i]  i += 1  k += 1    while j < R.length:  A[k] = Right[j]  j += 1  k += 1  return A |
| **Code** | def Merge\_Sort(A, l, r):  # BASE CASE  if l < r:  m = int(l + (r-l)/2)  Merge\_Sort(A, l, m)  Merge\_Sort(A, m+1, r)  Merge(A, l, m, r)  def Merge(A, l, m, r):  global S  size\_1 = m - l + 1 # size of left array  size\_2 = r - m # size of right array  # temporary arrays  L = []  R = []  # copying elements in temporary arrays  # and checkng if element < 0  for i in range(0, size\_1):  if A[l + i] <= 0:  L.append(A[l + i])  else:  R.append(A[l + i])  for j in range(0, size\_2):  if A[m + 1 + j] <= 0:  L.append(A[m + 1 + j])  else:  R.append(A[m + 1 + j])  # indexes of temp arrays  i = 0  j = 0  k = l  # copying in first Array  while i < len(L):  A[k] = L[i]  i += 1  k += 1  while j < len(R):  A[k] = R[j]  j += 1  k += 1 |
| **Time Complexity** | Best-case time complexity: O(nlgn)  Worst-case time complexity: O(nlgn) |
| **Proof of Correctness** | **Initialization:** Subarray S is empty. L[1] is the smallest element of L, while R[1] is the smallest element of R. So it is true.  **Maintenance:** if L[i] ≤ R[j]. Then L[i] is the smallest element not yet copied to S. The current nonempty part of S consists of the k − 1 smallest elements, so after the loop is over and L[i] is copied to S, the nonempty part of S will consist of the k smallest elements. Incrementing k (in the for loop update) and i re-establishes the loop invariant for the next iteration.  **Termination:** when k = m + 1 (Condition where loop terminates). S contains the m smallest elements of L and R, in sorted order. |
| **Advantages** | * It can sort larger input runtime complexity is far more less than previous algorithms O(nlgn) which is same for best and worst cases. * It is good for real life applications. * Its execution is sequential. It doesn’t require much seeking. |
| **Disadvantages** | * Its less efficient for small number of input. * It requires memory overhead for sub-array creation. * Even if array is already sorted then it’ll take same time that makes it worse O(nlgn) |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Quick Sort** |  |
| **Description** | Quick sort uses Divide & Conquer approach like Merge sort.  It picks a pivot and partitions array around the pivot, in such a way that lesser elements are moved to left and larger are moved to right.  we can pick first, last, random, or median as pivot  partitioning the array is main procedure.  partition is done so element as pivot is at its correct position and put all smaller elements to left and put all greater elements to right  it must take linear time.  It is “**Unstable**”. |
| **Pseudo Code** | Quick-Sort(A, lowest, highest)  // base case if single element then do nothing  if lowest >= highest  return  pivotIdx = Partition(A, lowest, highest)  Quick-Sort(A, lowest, pivotIdx – 1)  Quick-Sort(A, pivotIdx + 1, highest)  Partition(A, first, last)  A[PivotIdx] = A[last]  PivotIdx = first  loop idx from first to last  if A[idx] <= A[PivotIdx]  Swap(A[idx],A[PivotIdx])  pivotIdx = PivotIdx + 1  return PivotIdx – 1 |
| **Code** | def quick\_Sort(A, l, h):  if (l < h):  pi = partition(A, l, h)  quick\_Sort(A, l, pi - 1)  quick\_Sort(A, pi + 1, h)  def partition (A, l, h):  pivot = A[h];  i = (l - 1)  for j in range(l,h):  if (A[j] < pivot):  i+=1  A[i],A[j]=A[j],A[i]  A[i + 1],A[h]=A[h],A[i + 1]  return (i + 1) |
| **Time Complexity** | Best-case time complexity: O(nlgn)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** | **Basic Step:** when l < k since it contains one or no element so it is sorted trivally.  **Induction Step:** we take middle point as pivot and then element less than pivot are in L[] and Greater are in R[] when recursion occurs further R[i] and L[i] are sorted based on pivots and further sub-Divisions. Thus at each recursion pivot is at its rightful place.  Pivot P moves almost every element to its place in rightful manner till array is sorted. |
| **Advantages** | * Its runtime is O(nlgn) which is good for many practical applications and is widely used. * It consumes minimal memory overhead. * Its faster than Merge sort in average case. |
| **Disadvantages** | * Its runtime may vary depending upon the initial array. * Worst-case (Reverse Sorted Array) may cause O(n2) runtime complexity. * It’s not stable that may question its performance issues. |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Shell Sort** |  |
| **Description** | This algorithm is kin of insertion sort but it uses gaps and minimizes the comparisons.  It first calculate gap like (k = length) k/2, k/4,....,1  then comparison starts from the beginning of the list and after gap it changes positions of elements if they are not in the right order. This continues until element on the left is smaller than current element then gap is further divided into half and process is repeated. |
| **Pseudo Code** | SHELL-SORT(A,n)  gap = N/2, N/4, N/8, ...., 1  for gap=n/2, gap=0, gap/=2  //Perform gapped insertion sort for this gap size.    for i=gap to n with increment of 1  temp=A[i]    for j=i till j>=gap and A[j-gap]>temp    A[j]= A[j-gap]  j = j - gap  // put temp in its correct location  A[j]= temp; |
| **Code** | def Shell\_Sort(A):  n = len(A)  gap = n/2  while gap > 0:  for i in range(gap,n):  Temp = A[i]  j = i  while j >= gap and A[j-gap] > Temp:  arr[j] = arr[j-gap]  j -= gap  arr[j] = Temp  gap /= 2 |
| **Time Complexity** | Best-case time complexity: O(n \* lgn)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** |  |
| **Advantages** | * Requires no extra memory. * Faster than other n-squared algorithms like bubble sort. * Effective for finite numbers in an array. |
| **Disadvantages** | * It is Unstable. * Dependent on input array that hinders its performance. * Complex and difficult for understanding. |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Cocktail Sort** |  |
| **Description** | It is just like bubble sort but a n improved version. It is “**Stable**” algorithm.  It has two stages   * Left to Right: loop starts from left. Adjacent items are compared and if left value is greater than Right value, then values are swapped. At the end of first iteration, largest number will reside at the end of the array. * Right to Left: starting from the item before sorted item, to start of the array. Here adjacent items are compared and are swapped if required. |
| **Pseudo Code** | Cocktail-Sort(A)  swap = false  start = 0  end = A.length - 1  while (swap is True):  swap = False // in case array is already sorted  for i in range(start, end):  if (A[i] > A[i + 1]):  then swap( A[ i ], A[ i+1 ] )  swap := true;  if swapped = false then break  swap = false  end = end-1  for i in range(end, start-1, -1): // downto loop  if (A[i] > A[i + 1]):  swap (A[i + 1], A[i])  swap = True  start = start + 1 |
| **Code** | def cocktailSort(A):  n = len(A)  swap = True  start = 0  end = n-1  while (swap == True):  swap = False #in case array is already sorted    # Left to Right  for i in range(start, end):  if (A[i] > A[i + 1]):  A[i], A[i + 1] = A[i + 1], A[i]  swapped = True  if (swap == False):  break  swap = False  end = end-1  for i in range(end-1, start-1, -1):  if (A[i] > A[i + 1]):  A[i], A[i + 1] = A[i + 1], A[i]  swap = True  start = start + 1 |
| **Time Complexity** | Best-case time complexity: O(n)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** |  |
| **Advantages** | * It is faster than bubble sort. * It requires no additional space. * Best for small input. |
| **Disadvantages** | * No better than bubble sort when input is larger. * It is less efficient than algorithms with nlgn time. * It may cost time in order to save memory. |
| **Dry Run** | Description |

## **Searching:**

User can search entities following procedure facilitates in multicolumn composite filters

1. A search bar will be available for the query that’ll be searched initially from “**Title**” column.
2. If float is written after symbol “@” then it will be considered as Ratings and matching result will be shown from “**Ratings**” column.
3. String after “#” symbol will be considered as Genre matching result will be shown from “**Genre**” column.
4. A dropdown list will be available with **Column** name present in scrapped data and search will be commenced in selected column.
5. “**Starts with**” & “**Ends with**” provides entries starting with or ending with string within selected column.
6. AND, OR, NOT are given as radio button user will select a column and keyword user will be required to select one radio button it will generate a sub-query with multiple columns when he’s done simple press enter and search multi-columns.

Suppose:

“Action AND 2020 NOT 5” it’ll return all movies of action genre released in 2020 whose rating isn’t 5.

# **UI**

## **Wireframes:**

Following are some prototypes which were made in Pencil Tool pre-integration.

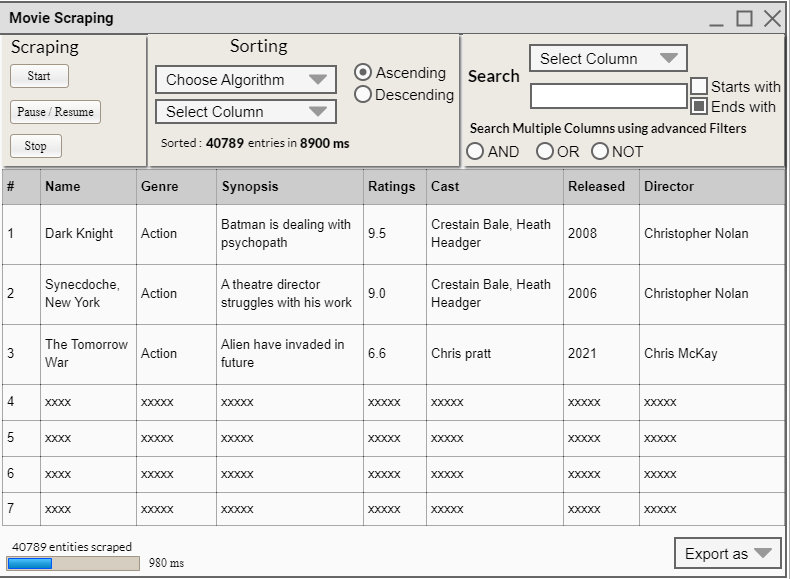


Fig 2: Prototype GUI made in Pencil tool.

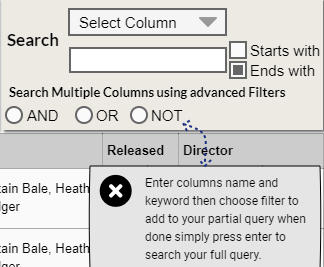
****

Fig 3: Searching for Composite filters

## **Implemented UI:**

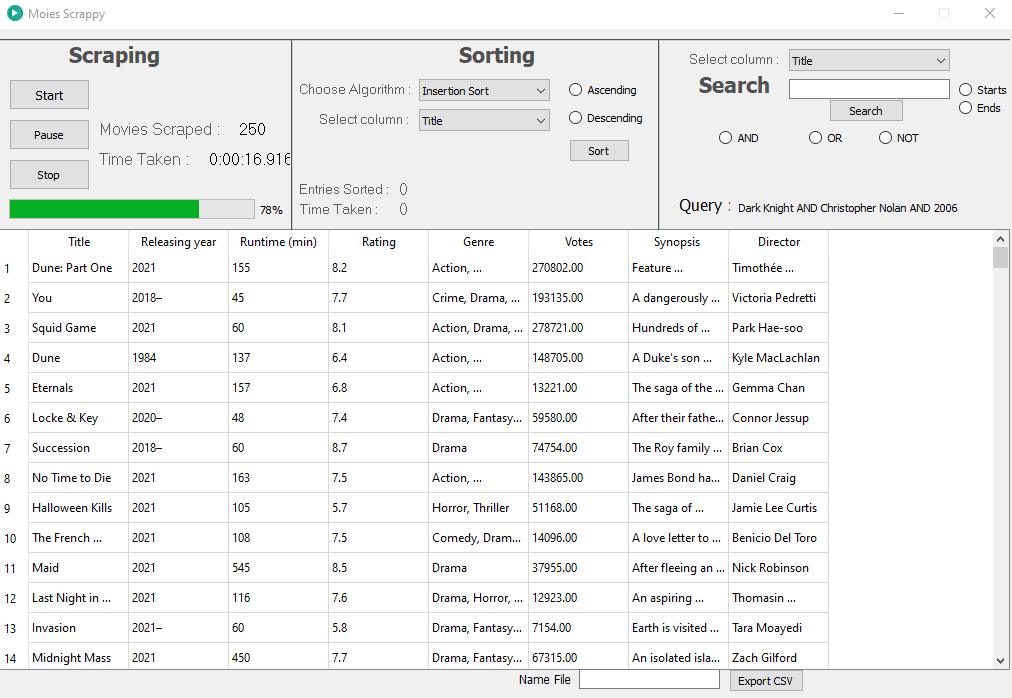


Fig 4: Scraping

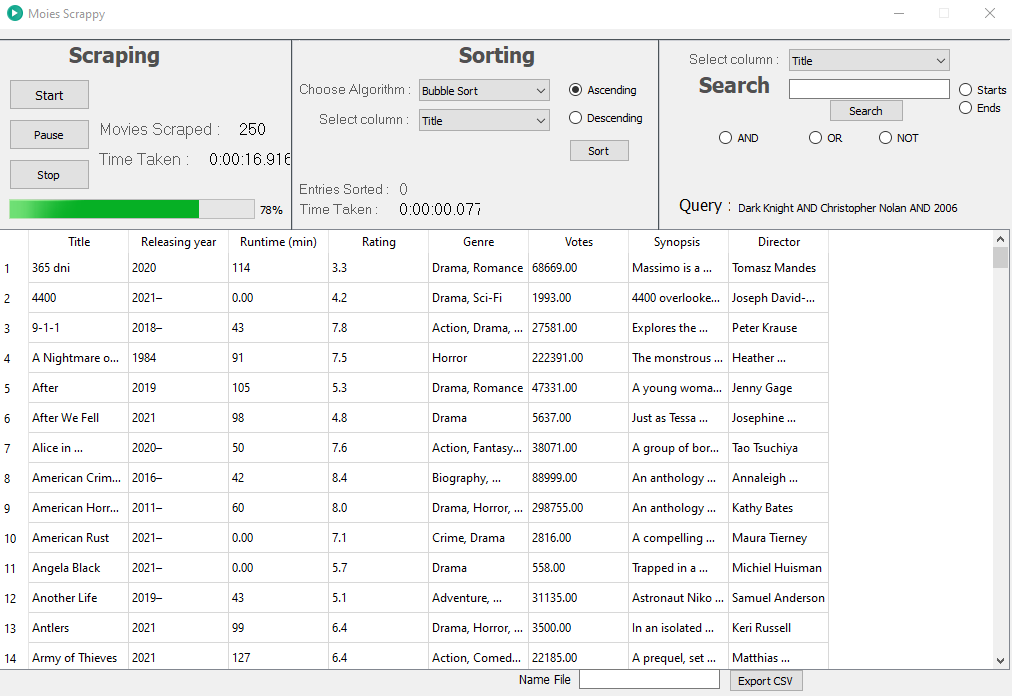


Fig 5: Sorting



Fig 6: Searching.

## **UI Components:**

|  |  |  |
| --- | --- | --- |
| **Component Name** | **Type** | **Purpose** |
| Scraping Panel | Panel | Holds buttons for scraping options |
| Start | Button | Start scraping the site |
| Pause / Resume | Button | Pauses and Resumes the Scraping |
| Sorting Panel | Panel | Holds options for sorting |
| Select Column | Combo Box | Avails name of different columns presented. |
| Select Algorithms | Combo Box | Avails name of different Sorting Algorithms.   * Selection Sort * Bubble Sort * Insertion Sort * Merge Sort * Quick Sort * Shell Sort * Cocktail Sort * Brick Sort |
| Ascending | Radio Button | Sorts data in ascending order |
| Descending | Radio Button | Sorts data in Descending order |
| Sorted | Label | Shows number of entries sorted and Time consumed during this process. |
| Search Panel | Panel | Holds options for searching |
| Select Column 2 | Combo Box | Avails name of different columns presented for commencing search. |
| Input box | Text Input | Enter keyword to search |
| Starts with | Checkbox | Search using keyword as starting reference. |
| Ends with | Checkbox | Search using keyword as Ending reference. |
| AND | Radio Button | Add AND operator to main Query |
| OR | Radio Button | Add OR operator to main Query |
| NOT | Radio Button | Add NOT operator to main Query |
| Table | Table | It’ll show data that’s point of interest.   * Name * Releasing Date * Runtime * Genre * Synopsis * Ratings * Votes * Director |
| Progress | Progress bar | shows how much data is scraped |
| Data scraped | Label | Amount of data scraped |
| Time | Label | Total time consumed during scraping |
| Export As | Combo Box | Option to export data file as CSV |

## **Features**

### **Scraping**

* Data source is remote site IMDB.
* Start Button: It scrapes by requesting html from URL which is modified to new page after previous is scraped so there’s minimal threat for getting IP blocked.
* Pause: stops scraping till pressed again.
* Stop: Stop scraping permanently.

### **Sorting**

Sorting is it’s another main feature.

* Algorithm Selection: Provides 8 different algorithms to sort.
* Column Selection: Select any column.
* Ascending, Descending Radios: Provide order of selection.
* Label: Count time taken for sorting.

### **Searching**

Searching provides following functionalities:

* Line Edit: Enter keyword to search
* Order: Search based on Start with or ends with

# **Problems**

Faced Following difficulties with project

|  |  |
| --- | --- |
| Concepts | I had dispersed ideas about algorithms in beginning. |
| Scraping | Scraping was a whole new thing it had to be mastered first. |
| Colleague | Colleague didn’t respond to any meeting and left me with all work till now so I had to do it individually. |
| QT designer | I had used visual studio so QT designer was a new thing but it caused less problem as it was similar to VS still it took two days to understand it’s functionalities |
| Implementation | Implementing GUI was less problem than adding functionalities o |
|  |  |
|  |  |

|  |  |
| --- | --- |
| Github Repository Link | [Umair-Manzoor-47/CS261F21PID07 (github.com)](https://github.com/Umair-Manzoor-47/CS261F21PID07) |