**CAB 301: Algorithms and Complexity**

**Assignment 1**

**Report and Analysis**

Name : Yonathan Cahyadi

Student Number : 10149953

# Algorithm for “Display the top 10 most frequently borrowed movie DVD”

For this function I decided to use *Merge Sort* as the sorting algorithm, the reason is *Merge Sort* works on any size of array and have a stable execution speed. How *Merge Sort* works, according to Wikipedia, it works by using divide and conquer technique. The first step is divide the unsorted array into *n* number of sub-array containing only *one* element, then repeatedly merge those sub-array using the specified condition ( in this case “how frequent the Movie is Borrowed”) until there is only one array left. This will be the sorted array.

## Pseudocode

### MergeByFrequencyOrder

**ALGORITHM** MergeByFrequencyOrder*(L[i…j], R[c…d])*

// Merge and sort the 2 sub-array *L[i…j]* and *R[c…d]* into a single sorted array *T[i…d]*

// The sorting is based on “How many times the Movie has been rented”

// initialize the array X to store the result of the merge

*resultLength* 🡨 *L.Length + R.Length*

*T* 🡨 *T[resultLength]*

// initialize the *index* for array *A, B* and *T*

*leftIndex* 🡨 *0* // index for array *L*

*rightIndex* 🡨 *0* // index for array *R*

*resultindex* 🡨 *0* // index for array *T*

// iterate through all the element in array *A* and *B*

**while** *leftIndex < L.Length* **and** *rightIndex**<**R.Length* **do**

// Check if both array had an element

**if** *leftIndex < L.Length* **and** *rightIndex < R.Length*

// Sort the element by how many times it has been rented

// if the *L* at *leftIndex* has been rented **less** **than or equal** *R* at *rightIndex*

**if** *L[leftIndex].timesRented <= R[rightIndex].timesRented*

// put the element of array *L* at *leftIndex* into array *T* at *resultindex*

*T[resultindex]* 🡨 *L[leftIndex]*

*leftIndex* 🡨 *leftIndex + 1*

*resultindex* 🡨 *resultindex + 1*

**else** //array *L* at *leftIndex* has been rented **more than** *R* at *rightIndex*

// put the element of array *R* at *rightIndex* into array *T* at *resultindex*

*T[resultindex]* 🡨 *R[rightIndex]*

*rightIndex* 🡨 *rightIndex + 1*

*resultindex* 🡨 *resultindex + 1*

**else if** *leftIndex < L.Length* // put the remaining element of array L into array T

*T[resultindex]* 🡨 *L[leftIndex]*

*leftIndex* 🡨 *leftIndex + 1*

*resultindex* 🡨 *resultindex + 1*

**else if** *rightIndex <**R.Length*// put the remaining element of array R into array T

*T[resultindex]* 🡨 *R[rightIndex]*

*rightIndex* 🡨 *rightIndex + 1*

*resultindex* 🡨 *resultindex + 1*

**return** *T*

### GetSortedMovieByFrequencyOrder

**ALGORITHM** *GetSortedMovieByFrequencyOrder(X[i…d])*

// Divide the array X into sub-array and do the Merge and Sorting algorithm

// The result will be stored in array T

**if** *X.Length <= 1* // if the array X only had one element return the array

**return** *X*

// get the mid point of the array X

*midPoint* 🡨 *X.Length / 2*

// initialize array to store the Result

*T* 🡨 *T[X.Length]*

// initialize 2 sub-array L and R

*L* 🡨 *L[midPoint]*

**If** *X.Length % 2 == 0* // if the Length of array X is **even**

*R* 🡨 *R[midPoint]*

**else** // if the Length of array X is **odd**

*R* 🡨 *R[midPoint + 1]*

// populate the left array using elements from X[i…midPoint]

**for** *index* 🡨 *0* **to** *midPoint* **do**

*L[index]* 🡨 *X[index]*

// populate the right array using elements from X[midPoint…d]

*rightSubArrayIndex* 🡨 *0*

**for** *index* 🡨 *midPoint* **to** *X.Length* **do**

*R[rightSubArrayIndex]* 🡨 *X[index]*

*rightSubArrayIndex* 🡨 *+ 1*

// recursively divide the array until there is 1 element left

*L* 🡨 *GetSortedMovieByFrequencyOrder(L)*

*R* 🡨 *GetSortedMovieByFrequencyOrder(R)*

// merge L and R into one sorted array

*T 🡨 MergeByFrequencyOrder(L, R)*

**return** *T*

### AddToArray

**ALGORITHM** *AddToArray(BinaryTreeNode, A[0…numberOfRegisteredDVD], index)*

// This algorithm will traverse through all the Node in Binary Tree and transfer all the Node // into an array A[0…numberOfRegisteredDVD]

**if** *BinaryTreeNode == null* // if the current node is empty return the index

**return** *index*

*A[index] 🡨 BinaryTreeNode.data*

*index 🡨 index + 1*

// Traverse through all the Binary Tree Node and put the data into the array A

**if** *BinaryTreeNode.left != null*

*index 🡨 AddToArray(BinaryTreeNode.left, A, index)*

**if** *BinaryTreeNode.right != null*

*index 🡨 AddToArray(BinaryTreeNode.right, A, index)*

**return** *index*

### GetAllMovie

**ALGORITHM** *GetAllMovie()*

// This algorithm will get all the Movie from Movie Collection Binary Tree

**if** *binaryTreeRoot == null*

**return** *null*

*R 🡨 R[numberOfRegisteredDVD]* // Initialize array with size of Registered DVD

*AddToArray(binaryTree, R, 0)*

**return** *R*

### MemberGetTop10

**ALGORITHM** *MemberGetTop10()*

// This will display the Top 10 most borrowed DVD in descending order

*A 🡨 GetAllMovie()* // Get array containing all registered the Movie

**if** *A != null* // Check if there is any movie registered in movie collection

*T 🡨 GetSortedMovieByFrequencyOrder(A)* // Get the sorted version of array A

// iterate through all the array T

**for** *index 🡨 T.Length – 10* **to** *T.Length* **do**

// Check if the index is bigger or equals to 0

// To avoid index out of bound

**if** *index >= 0*

**output** *T[index].title*

**output** *T[index].starring*

**output** *T[index].director*

**output** *T[index].genre*

**output** *T[index].duration*

**output** *T[index].releaseDate*

**output** *T[index].classification*

**output** *T[index].timesRented*

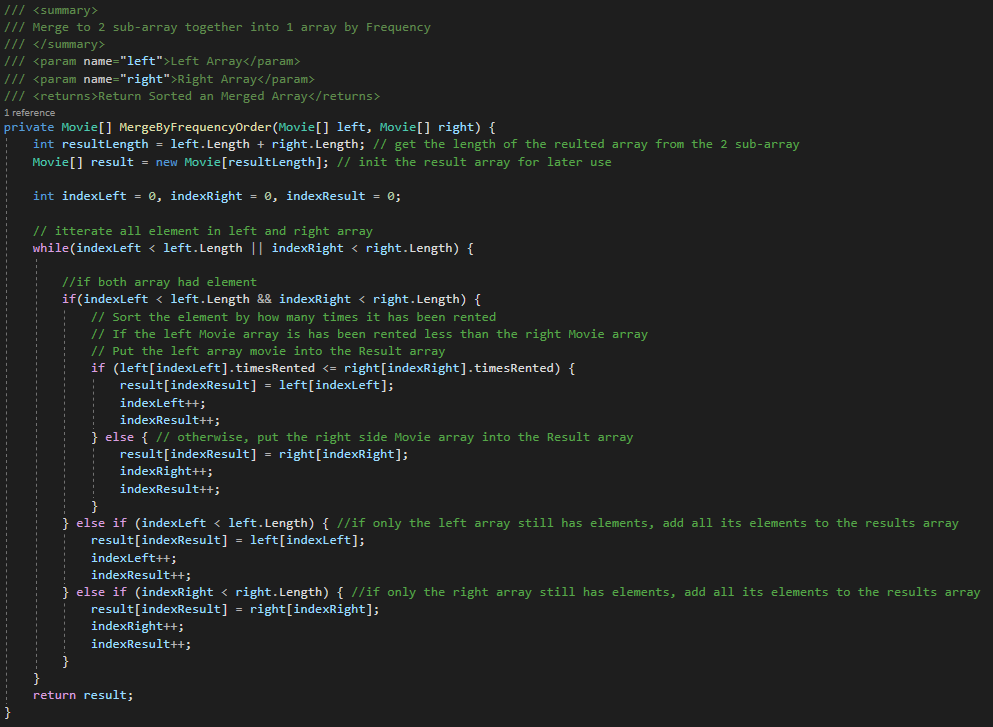
*index 🡨 index + 1*

**else**

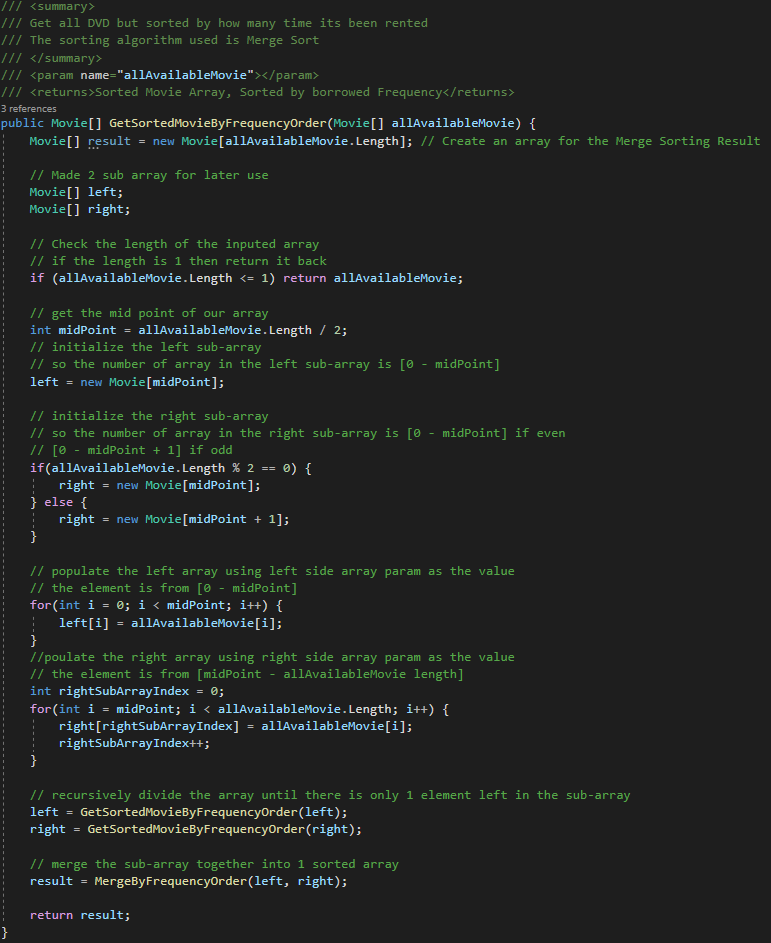
**output** “*No DVD Registered”*

## Code Implementation

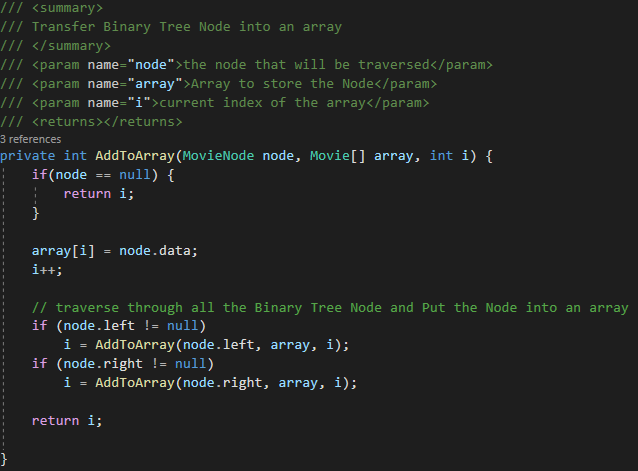
### MergeByFrequencyOrder



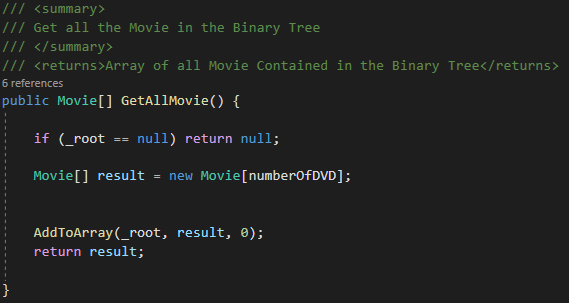
### GetSortedMovieByFrequencyOrder



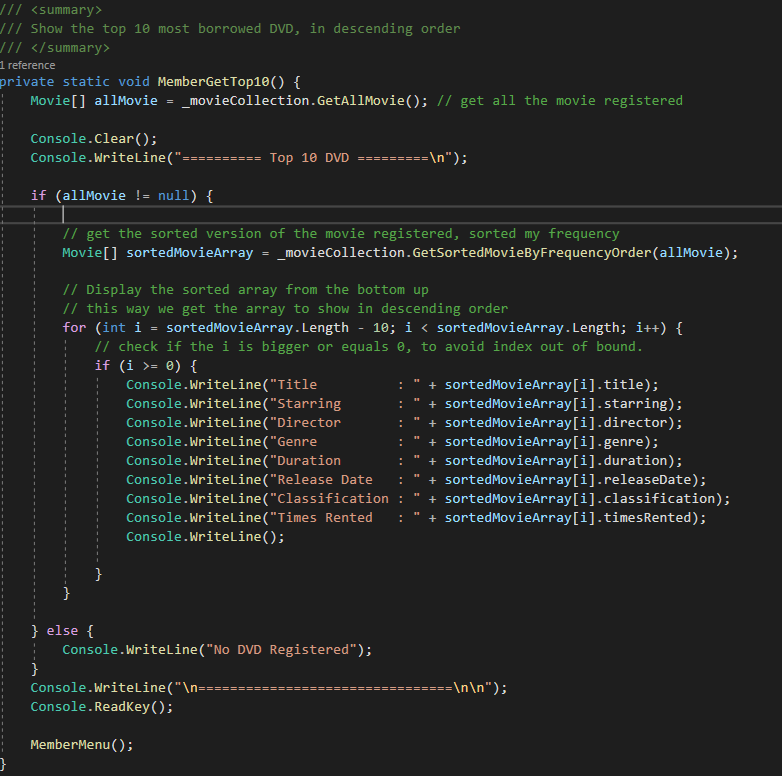
### AddToArray



### GetAllMovie



### MemberGetTop10



# ­Analysis of the Time Complexity

## In Theory

A diagram with a tree on the left and merging times on the right. The tree is labeled "Subproblem size" and the right is labeled "Total merging time for all subproblems of this size."
The first level of the tree shows a single node n and corresponding merging time of c times n. The second level of the tree shows two nodes, each of 1/2 n, and a merging time of 2 times c times 1/2 n, the same as c times n. The third level of the tree shows four nodes, each of 1/4 n, and a merging time of 4 times c times 1/4 n, the same as c times n. The fourth level of the tree shows eight nodes, each of 1/8 n, and a merging time of 8 times c times 1/8 n, the same as c times n. Underneath that level, dots are shown to indicate the tree continues like that. A final level is shown with n nodes of 1, and a merging time of n times c, the same as c times n.

Time per Level

Level *n*

Level 8

Level 4

Level 2

Level 1

[Merge Sort Visualization]. https://cdn.kastatic.org/ka-perseus-images/5fcbebf66560d8fc490de2a0d8a0e5b1d65c5c54.png

From the images above we can see that merge sort works by dividing each problem into a smaller sub-problem. The sub-problem will be divided until there is only element left, and every time the sub-problem is divided it will take a ***time proportional to its number of element*** ***left*** in the sub-array so the time complexity needed per level to divide the sub-problem into smaller ploblem is . In addition, the number of level needed to divide a problem size into problem size is unknown, so we can assume it need to ***keep dividing every sub-problem elements in halves*** until it reach where every sub-problem have a size of 1. Therefore, we can tell, that the time complexity needed by the algorithm to reach where every sub-problem has a size of 1 is . So the total time complexity needed for merge sort to finish sorting is:

and we know that,

.

Therefore we can do:

Which is

For **best, average** and **worst-case time** complexity.

In Graph it will be shown as,

A screenshot of a cell phone

Description automatically generated

## Implementation

### MergeByFrequenceyOrder

|  |  |
| --- | --- |
| int resultLength = left.Length + right.Length; |  |
| Movie[] result = new Movie[resultLength]; |  |
| int indexLeft = 0, indexRight = 0, indexResult = 0; |  |
|  |  |
| while(indexLeft < left.Length || indexRight < right.Length) |  |
| if(indexLeft < left.Length && indexRight < right.Length) |  |
| if (left[indexLeft].timesRented <= right[indexRight].timesRented) |  |
| result[indexResult] = left[indexLeft]; |  |
| indexLeft++; |  |
| indexResult++; |  |
| else |  |
| result[indexResult] = right[indexRight]; |  |
| indexRight++ |  |
| indexResult++; |  |
| else if (indexLeft < left.Length) |  |
| result[indexResult] = left[indexLeft]; |  |
| indexLeft++; |  |
| indexResult++; |  |
| else if (indexRight < right.Length) |  |
| result[indexResult] = right[indexRight]; |  |
| indexRight++; |  |
| indexResult++; |  |
|  |  |
| return result; |  |

From the table we can see that the time complexity of this function is .

### GetSortedMovieByFrequencyOrder

|  |  |
| --- | --- |
| Movie[] result = new Movie[allAvailableMovie.Length]; |  |
| Movie[] left; |  |
| Movie[] right; |  |
|  |  |
| if (allAvailableMovie.Length <= 1) |  |
| return allAvailableMovie; |  |
|  |  |
| int midPoint = allAvailableMovie.Length / 2; |  |
| left = new Movie[midPoint]; |  |
|  |  |
| if(allAvailableMovie.Length % 2 == 0) |  |
| right = new Movie[midPoint]; |  |
| else |  |
| right = new Movie[midPoint + 1]; |  |
|  |  |
| for(int i = 0; i < midPoint; i++) |  |
| left[i] = allAvailableMovie[i]; |  |
|  |  |
| int rightSubArrayIndex = 0; |  |
| for(int i = midPoint; i < allAvailableMovie.Length; i++) |  |
| right[rightSubArrayIndex] = allAvailableMovie[i]; |  |
| rightSubArrayIndex++; |  |
|  |  |
| left = GetSortedMovieByFrequencyOrder(left); |  |
| right = GetSortedMovieByFrequencyOrder(right); |  |
| result = MergeByFrequencyOrder(left, right); |  |
| return result; |  |

From this table we can see that the time complexity of this function is .

### AddToArray

|  |  |
| --- | --- |
| if(node == null) |  |
| return i; |  |
|  |  |
| array[i] = node.data; |  |
| i++; |  |
|  |  |
| if (node.left != null) |  |
| i = AddToArray(node.left, array, i); |  |
| if (node.right != null) |  |
| i = AddToArray(node.right, array, i); |  |
|  |  |
| return i; |  |

From this table we can see that the time complexity of this function is .

### GetAllMovie

|  |  |
| --- | --- |
| if (\_root == null) |  |
| return null; |  |
|  |  |
| Movie[] result = new Movie[numberOfDVD]; |  |
|  |  |
| AddToArray(\_root, result, 0); |  |
|  |  |
| return result; |  |

From this table we can see that the time complexity of this function is

### MemberGetTop10

|  |  |
| --- | --- |
| Movie[] allMovie = \_movieCollection.GetAllMovie(); |  |
|  |  |
| Console.Clear(); |  |
| Console.WriteLine("========== Top 10 DVD =========\n"); |  |
|  |  |
| if (allMovie != null) |  |
| Movie[] sortedMovieArray = \_movieCollection.GetSortedMovieByFrequencyOrder(allMovie); |  |
| if (i >= 0) |  |
| Console.WriteLine("\tTitle : " + sortedMovieArray[i].title); |  |
| Console.WriteLine("\tStarring : " + sortedMovieArray[i].starring); |  |
| Console.WriteLine("\tDirector : " + sortedMovieArray[i].director); |  |
| Console.WriteLine("\tGenre : " + sortedMovieArray[i].genre); |  |
| Console.WriteLine("\tDuration : " + sortedMovieArray[i].duration); |  |
| Console.WriteLine("\tRelease Date : " + sortedMovieArray[i].releaseDate); |  |
| Console.WriteLine("\tClassification : " + sortedMovieArray[i].classification); |  |
| Console.WriteLine("\tTimes Rented : " + sortedMovieArray[i].timesRented); |  |
| Console.WriteLine(); |  |
| else |  |
| Alert("No DVD Registered"); |  |
|  |  |
| Console.WriteLine("\n================================\n\n"); |  |
| Console.ReadKey(); |  |
|  |  |
| MemberMenu(); |  |

From this table we can see that this function time complexity is.

### The Conclusion

From all the analysis above, we know that the function needed to get the top 10 most borrowed DVD have a Time Complexity of **.** In a graph it will be shown as:

A screenshot of a cell phone

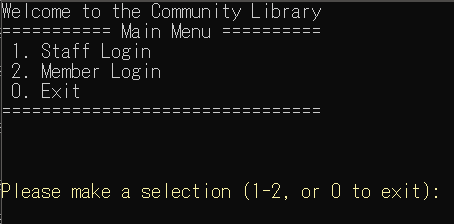
Description automatically generated

The graph looks almost the same with the graph discussed in the Theory Section [2.1]. Therefore, I can conclude that my Merge Sort implementation is correct and up to the expected time efficiency.

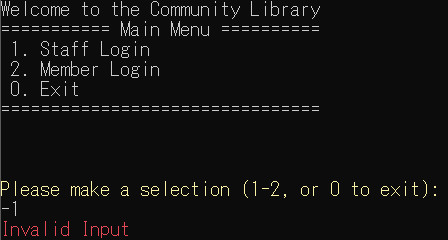
# Screenshots of each Functional test

## Main Menu

### Main Menu Selection

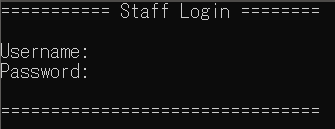


### Main Menu Invalid Selection

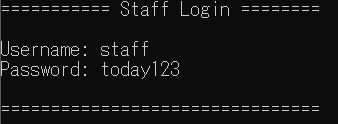


## Staff Login

### Staff Login

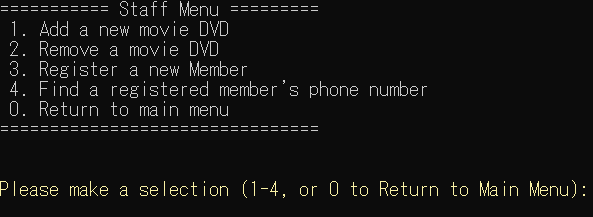


### Staff Login with Username and Password

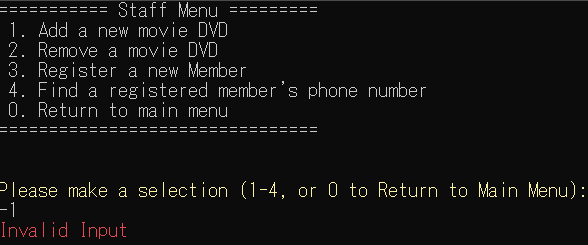


## Staff Menu

### Staff Menu Selection



### Staff Menu Invalid Selection



## Add a new movie DVD

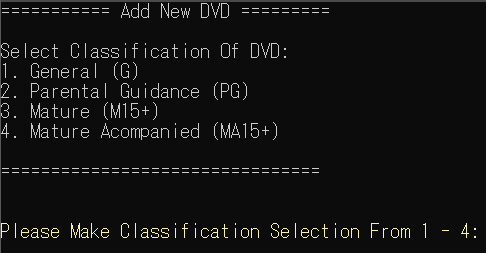
### Add new DVD Filling the DVD Title, Director, Starring, Release Date and Duration



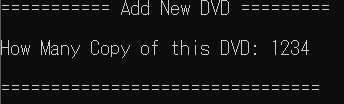
### Select DVD Genre



### Select DVD Classification



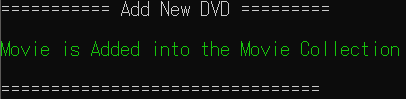
### Add Number of Available DVD Copy



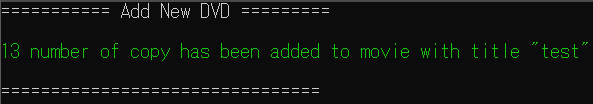
### Add DVD into the Movie Collection



### DVD Added to movie Collection Successfully



### 2.4.7 DVD With the Same Title in the Movie Collection is added

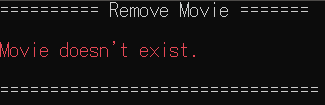


## Remove a movie DVD

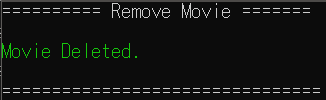
### Remove DVD



### DVD is Not in the Collection



### DVD is Successfully Removed

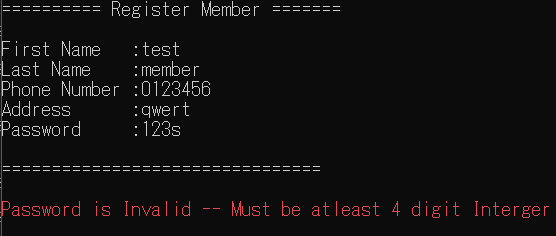


## Register New Member

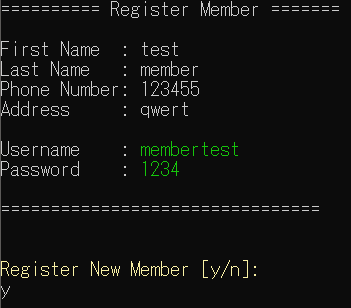
### Add New Member



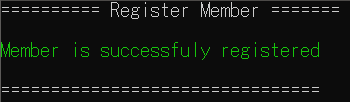
### Password Less than 4 digit or Not an Interger



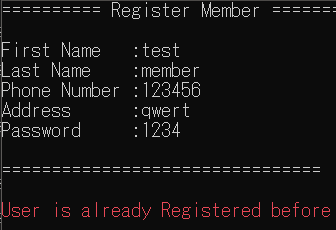
### Adding New User into Member Collection and Generating Member Username



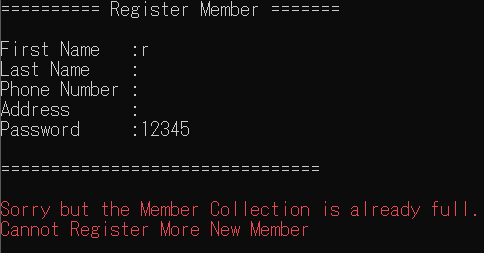
### New Member is Successfully Registered and added into the Member Collection



### Member is Already Registered Before

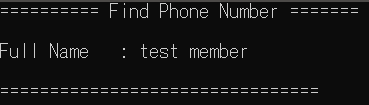


### Member Collection is Already Full

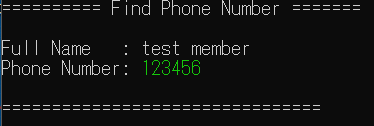


## Find a Registered Member’s Phone Number

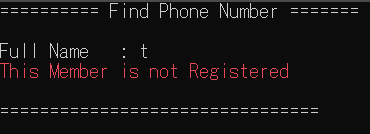
### Find Member Phone Number



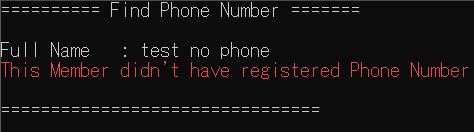
### User Founded



### User is Not Registered

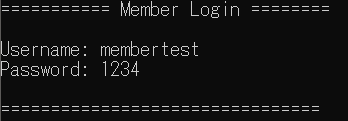


### User didn’t have Registered Phone Number



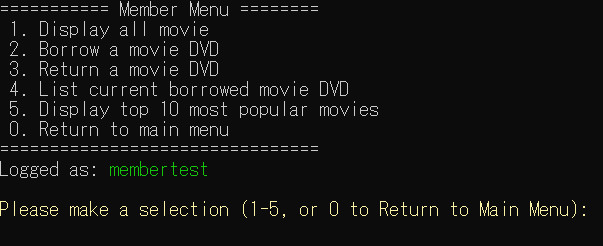
## Member Login

### Member Login

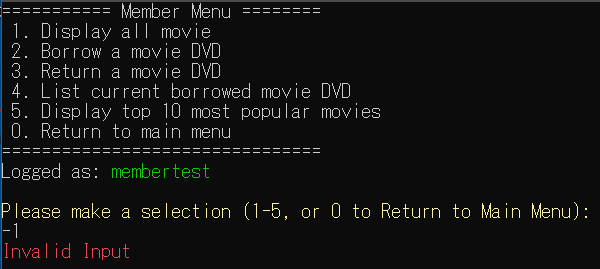


## Member Menu

### Member Menu Selection

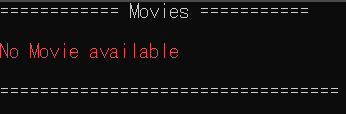


### Member Menu Invalid Selection

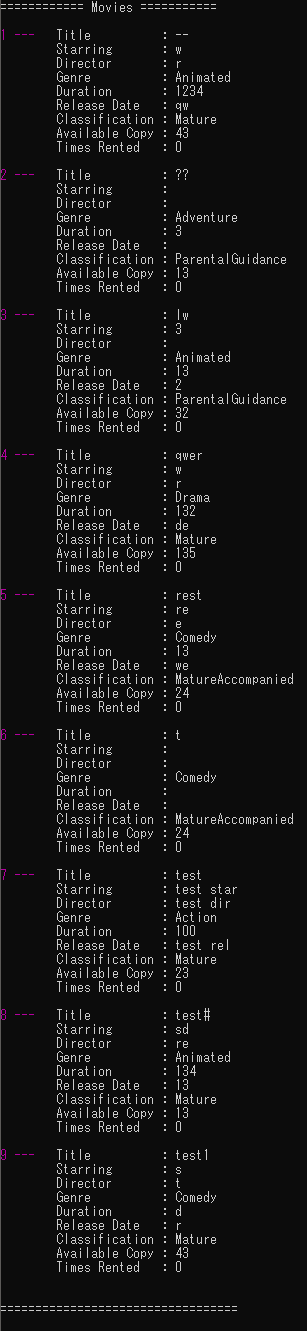


## Display All Movie

### No Movie Registered

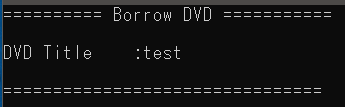


### Display All Registered Movie



## Borrow a Movie

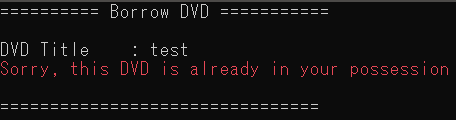
### Borrow a Movie



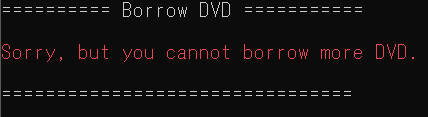
### Movie is Not Available for Renting



### Movie is Already in Possession

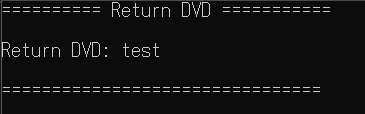


### Member Borrowing Limit Already Reached

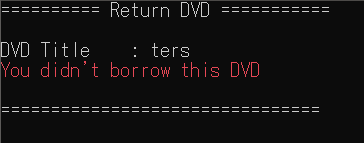


## Return a Movie

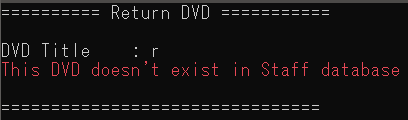
### Return a Movie



### Movie is Not Borrowed

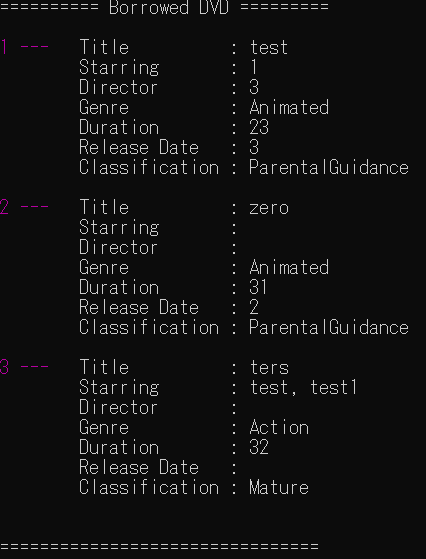


### DVD is Deleted by Staff, but Still in Member Possession

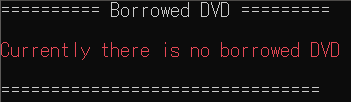


## List All Borrowed Movie DVD

### List All Borrowed Movie DVD

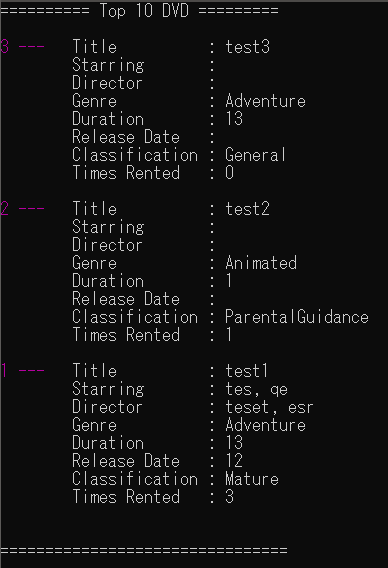


### No DVD Borrowed



## Display top 10 Most Popular Movie

### List of top 10 Most Popular Movie in Descending Order



### No Movie is Registered



# Reference

Wikipedia. *Merge Sort*. Retrieved May 22, 2020, from <https://en.wikipedia.org/wiki/Merge_sort>