## Algorithm for displaying top 10 most frequently borrowed movies

The movie collection binary search tree (BST) is flattened to an array with a traversal algorithm, and the array is then sorted with a quicksort sorting algorithm based on the movies’ borrowed count.

### Traversal algorithms used to flatten BST:

**ALGORITHM** GetArrayMovies()

// Flattens binary search tree to an array by inorder traversal

// Input: Global variables *root* which is the root node of the

// BST, an empty array *arrayMovies[]*, amount of BST

// nodes *moviesCount,* and *arrayMovies* index *arrayIndex*

// Output: Array *array[0…moviesCount-1]* filled with elements from BST

arrayMovies = new array[0…moviesCount-1] // create new empty array of size *moviesCount*

arrayIndex ←0

FlattenMovies(root)

**return** arrayMovies

**ALGORITHM** FlattenMovies(node)

// Traverses through BST nodes while adding each node to the global variable *arrayMovies*

// Input: Current BST node *node*, and global variables *arrayIndex* and *arrayMovies*

// Output: Updated *arrayMovies* with new node

**if** node != null **then**

FlattenMovies(node.left) // traverse down left subtree

arrayMovies[arrayIndex] ← node.movie

arrayIndex ← arrayIndex + 1

FlattenMovies(node.right) // traverse down right subtree

In the best, average, and worst-case scenario for flattening the BST, every node is visited once. Therefore, the time efficiency is:

Since a new array is created with its length the number of nodes, the space efficiency is:

### Quicksort sorting algorithm used to sort the array of movies by borrowed count:

**ALGORITHM** partition(arr[l..r], l, r)

// Partitions a subarray by Hoare's algorithm, using the first element

as a pivot

// Input: Subarray of array arr[0..n-1], defined by its left and right

// indices l and r (l < r)

// Ouput: Partition of arr[l..r], with the split position returned as

// the function's value

Movie p ← arr[l];

i ← l-1

j ← r+1

**repeat**

**repeat** i ← i + 1 **until** arr[i] ≤ p // compare amount borrowed

**repeat** j ← j – 1 **until** arr[j] ≥ p

**if** i ≥ j **then**

**return** j

swap(arr[i], arr[j])

**ALGORITHM** sort(arr[l..r], l, r)

// Sorts a subarray by quicksort

// Input: Subarray of array arr[0..n-1], defined by its left and right

// indices l and r

// Output: Subarray arr[l..r] sort in decreasing order

**if** l < r

s ← partition(arr, l, r) // split position

sort(arr, l, s)

sort(arr,s+1, r)

##### Worst-case-analysis

A worst-case scenario will occur when there is an unbalanced partition where one of the subarrays returned is of size *n* - 1. This may occur if the pivot is either the largest element (in our case, the most borrowed) or the smallest (least borrowed). If this happens for every partition, the recursive call will compute an array of size 1 less than the previous. Therefore, *n*-1 nested calls are made before reaching an array size of 1. The time complexity for this can be expressed as so:

This yields that:

##### Best-case analysis

A best-case scenario will occur if there are two equal size subarrays after each partition. This will cause nested calls before the reaching an array size of 1, where each call level only needs time. Therefore, the time complexity is:

**Average-case analysis**

An average-case scenario is similar to a best-case scenario, except the subarrays are not perfectly balanced after each partition. The difference of subarrays sizes after a partition approaches a medium of equality after each nested call. An average-case scenario would approach that of the best-case, rather than the worst case. The time complexity is:

**Overall Efficiency of the quicksort algorithm:**

Time efficiency:

* Worst case:

Space efficiency:

* No temporary storage needed

Not Stable