Complexity Analysis:

(B) BUBBLE SORT:

Temporal Complexity:

n = books.size()

```
public void bubbleSort(ArrayList<Book> books) {
        Book temp = null;                                                   1
        for(int i = books.size()-1;i>0;i--) {                               n
        for(int j = 0;j<i; j++) {                                           ((n(n+1))/2) + n
         if(books.get(j).getValue()> books.get(j+1).getValue()) {           ((n(n+1))/2) + 1
        temp = books.get(j);                                                ((n(n+1))/2) +1
        books.set(j, books.get(j+1));                                       ((n(n+1))/2) +1
        books.set(j+1, temp);                                               ((n(n+1))/2) +1
                                }
                        }
                }
        }
```

T(B) = 1 + (n) + ((n(n+1))/2) + n + ((n(n+1))/2) + 1 + ((n(n+1))/2) + 1 + ((n(n+1))/2) + 1 + ((n(n+1))/2) + 1

T(B) = (n) + ((n(n+1))/2) + n + ((n(n+1))/2) + ((n(n+1))/2) + ((n(n+1))/2) + ((n(n+1))/2) + 5

T(B) = n + ((5n^2 + 5n)/2) + 5

T(B) = ((5n^2 + 7n)/2) + 5

Temporal Complexity = O(n^2)

Merge Sort

Merge

Line	**Quantity of times that is repeated**
int leftIndex = 0;	1
int rightIndex = 0;	1
int listIndex = 0;	1
while(leftIndex < left.size() && rightIndex < right.size()) {	n+1
if(new ValueComparator().compare(left.get(leftIndex), right.get(rightIndex))<0)	n
list.set(listIndex, left.get(leftIndex));	n
leftIndex++;	n
}else {	n
list.set(listIndex, right.get(rightIndex));	n
rightIndex++;	n
listIndex++;	n
ArrayList<Book> temp;	1
int tempIndex = 0;	1
if(tempIndex>=left.size())	1
temp = right;	1
tempIndex = rightIndex;	1

else	1
temp = left;	1
tempIndex = leftIndex;	1
for(int i = tempIndex;i<temp.size();i++) {	n+1
list.set(listIndex, temp.get(i));	n
listIndex++;	n

T(n)= 11n+13

T(n)= O (n)

MergeSort

Line	**Quantity of times that is repeated**
ArrayList<Book> left = new ArrayList<Book>();	1
ArrayList<Book> right = new ArrayList<Book>();	1
int medium;	1
if(list.size() == 1) {	1
return list;	1
}else {	1
medium = list.size()/2;	1

for(int i = 0 ; i < medium; i++) {	n+1
left.add(list.get(i));	n
for(int i = medium; i<list.size(); i++) {	n+1
right.add(list.get(i));	n
left = mergeSort(left);	1
right = mergeSort(right);	1
merge(left, right, list);	O(n)
return list;	1

T(n)= O(n/2)+3n+12

At first glance we see that the merge ordering is O (n), however we must bear in mind that this growth is not the fastest and we must take other things into account, since the merge is based on the divide and conquer method, which What it does is divide a problem into subproblems giving a solution to this subproblem so that it solves the entire problem. So in this case we have to:

DC). Divide: At this point we take half of the arrangement, so this is n / 2, this division occurs in a Constant O (1) complexity.

V (c) You will win: here the solution is given to the subproblem that is given by n / 2 and being recursive it will be executed twice, therefore it will be given by: 2T (n / 2).

Combine: In this case it is O (n) because that is how it gave us our temporal analysis of the merge sort.

With those values we can get the following recurrence:

c = constant equal to the time required to solve the algorithm

when n = 1 then c

when n> 1 then 2T (n / 2) + cn

When checking this recurrence with different values of n we obtain:

n = 1, time = c

n = 2, time = 2T (1) + 2c = 2c + 2c = 4c = c (2 lg (2) + 2)

n = 4, time = 2T (2) + 4c = 8c + 4c = 12c = c (4 lg (4) + 4)

n = 6, time = 2T (3) + 6c = 18c + 6c = 24c = c (6 lg (6) + 6)

for which we see that:

n = n, time = 2T (n / 2) + nc = c (n lg (n) + n)

Therefore we can conclude that the merge sort algorithm has a complexity of O (n lg n).

HeapSort:

Hipify

Line	**Quantity of times that is repeated**
int largest = i;	n
int l = 2*i + 1;	n
int r = 2*i + 2;	n
if (l < n && books.get(l).getValue() > books.get(largest).getValue()) {	n
largest = l;	n
if (r < n && books.get(r).getValue() > books.get(largest).getValue()) {	n
largest = r;	n
if (largest != i){	n
Book swap = books.get(i);	n
books.set(i, books.get(largest));	n
books.set(largest, swap);	n
heapify(books, n, largest);	log n

T(n)= log n +12 n

T(n)= O(log n)

HeapSort

Line	**Quantity of times that is repeated**
int n = books.size();	1
for (int i = n / 2 - 1; i >= 0; i--) {	n+1
heapify(books, n, i);	n
for (int i=n-1; i>=0; i--){	n+1
Book temp = books.get(0);	n
books.set(0,books.get(i));	n
books.set(i, temp);	n
heapify(books, i, 0);	log n

T(n)= log n +6n+3

T(n)= O(n log n)

Space Complexity:

Type	Variable	Quantity of atomic values
Input	books	n
Auxiliary	temp i j	1 1 1
Output		

Total Space Complexity: n + 3

Space Complexity = ɵ (n)

HeapSort

Heapify

Type	Name	Atomic values quantity
Input	books n i	n 1 1
Auxiliary	swap	1
Output	-	-

Total Space Complexity: n+3

Space Complexity = ɵ (n)

HeapSort

Type	Name	Atomic values quantity
Input	books	n
Auxiliary	i temp	1 1
Output	-	-

Total Space Complexity: n + 3

Space Complexity = ɵ (n)

MergeSort

MergeSort

Type	Name	Atomic values quantity
Input	list	n

Auxiliary	i medium left right	1 1 n/2 n/2
Output	-	-

Total Space Complexity: 2n + 2

Space Complexity = ɵ (n)

Merge:

Type	Name	Atomic values quantity
Input	list right left	n 1 1
Auxiliary	i temp leftIndex rightIndex	1 1 n/2 n/2
Output	-	-

Total Space Complexity: 2n + 4

Space Complexity = ɵ (n)