```
function mergeSort(Array arr, Array tempArr, int first, int last) {
   mid = (first + last) / 2  // initial condition    k0

   if (fisrt < last) {
      mid = (first + last) / 2          k1
      mergeSort(arr, tempArr, first, mid)         log(n) / 2 calls
      mergeSort(arr, tempArr, mid + 1, last)      log(n) / 2 calls
   }
   merge(arr, tempArr, first, mid, last)      O(n), see below
}
```

Conclusion:
mergeSort() contains b calls, which 2^b = n, b = log(n) base 2, consider as O((k0 + k1) * logn) = O(logn), this is always true no matter the best case or the worst case.
worst case complexity = O(n * log(n)) = O(nlogn)

```
function merge(Array arr, Array tempArr, int first, int mid, int last) {
   // arr1 & arr2 means 2 part in the arr to be merged together
   int beginHalf1 = first;
 int endHalf1 = mid;
 int beginHalf2 = mid + 1;          k0
 int endHalf2 = last;
 int index = 0;

  while((beginHalf1 <= endHalf1) && (beginHalf2 <= endHalf2))    iterate 2 arrays -> O(2 * n) = O(n)
 {
    if(array[beginHalf1].compareTo(array[beginHalf2]) < 1)      k1
    {
     tempArray[index] = array[beginHalf1];
     beginHalf1++;
    }
    else             k2
    {
     tempArray[index] = array[beginHalf2];
     beginHalf2++;
    }
    index++;
 }

   // when arr1 is empty, but arr2 is not
   while((beginHalf1 >= endHalf1) && (beginHalf2 <= endHalf2))     Worst case: arr1 empty, arr2 full -> O(n)
 {
    tempArray[index] = array[beginHalf2];     k3
    beginHalf2++;
    index++;
 }

 // when the right part has completly copied to tempArray
   while((beginHalf2 >= endHalf2) && (beginHalf1 <= endHalf1))       Worst case: arr2 empty, arr1 full -> O(n)
      {
    tempArray[index] = array[beginHalf1];      k4
    beginHalf1++;
```

```
    index++;
 }

   index = 0       // copy back to the original
   for(int i = first; i <= last; i++)       iterate the whole array -> O(n)
 {
    if(tempArray[index] != null)        k5
  array[i] = tempArray[index];
    index++;
   }
}
```

O(k0 + 2n(k1 + k2) + n(k3) + n(k4) + n(k5)) = O(n)