6

a

1. F
2. T
3. F
4. T
5. T

b

```
merge (array[], left, mid, right){
    leftIndex from left to mid
    rightIndex from mid + 1 to right
    rightIndex move right until array [leftIndex] and array[rightIndex] are
identical or rightIndex out of range
    Then look at leftIndex.
    leftIndex move left until array [leftIndex] and array[rightIndex] are
identical or leftIndex out of range
    we will put the identical pairs in an array
}
divide group(array[], begin, end){
    if begin equals to end, return
    mid = (end + begin) / 2
    divide group(array, begin, mid)
    divide group(array, mid + 1, end)
    merge(array, begin, mid, end)
}
```

Correction: The way we run is same as merge sort, so we can get every pair, one from left and one from right. After comparison, we can get all of the identical pair and store it in an array.
Time Complexity: The algorithm we use is modified by merge sort, so the time complexity is $ O(n*logn) $. *Moreover, because we need to compare the segment, we need to spend O(K) time inside the merge. Thus, the time complexity is $O(n*logn*K)$*

c

```
compare merge (array[], left, mid, right, fstatus, bstatus){
    total = fstatus + bstatus
    if total is bigger than 1, stop comparing process
    else return total
}
compare merge(segment1[], segment2[], begin, end, status){
    if begin equals to end, compare segment1 and segment2
        if they are same return 0, else return 1
```

```
    mid = (end + begin) / 2
    fstatus = divide group(segment1[], segment2[], begin, mid)
    bstatus = divide group(segment1[], segment2[], mid + 1, end)
    status = compare merge(segment1[], segment2[], begin, mid, end, fstatus,
bstatus)
    return status
}
merge (array[], left, mid, right){
    leftIndex from left to mid
    rightIndex from mid + 1 to right
    rightIndex move right until array [leftIndex] and array[rightIndex] are
similar(call compare merge) or rightIndex out of range
    Then look at leftIndex.
    leftIndex move left until array [leftIndex] and array[rightIndex] are
similar(call compare merge) or leftIndex out of range
    we will put the similar pairs in an array
}
divide group(array[], begin, end){
    if begin equals to end, return
    mid = (end + begin) / 2
    divide group(array, begin, mid)
    divide group(array, mid + 1, end)
    merge(array, begin, mid, end)
}
```

d

Correctness: We do same as 6-(b), but we change the way we compare. We compare them by cutting the segment into to parts until there is only one element left. Then, we compare two segment if they are same, there is no difference. Then, we can count all the difference. Once, the difference is bigger than 1, we stop comparing. Thus, we can get all similar pairs of segment.

e

$f(a) = aloga, f(b) = blogb, f(a+b)=(a+b)log(a+b) $
$ f(a+b)=(a+b)log(a+b)=(a+b)(log*a**logb)*=*a*log*a**logb* + *b*log*a**logb$*
*$ a*log*a**logb* >= *a*loga$
$ b*loga*logb >= blogb$
Thus,
$ f(a) + f(b) <= f(a+b) $

f

When comparing, we use the algorithm similar to merge sort, when we merge, we only use $O(1)$ time. Thus, in fact we only use at most O(K) time to compare. So, our time complexity is $O(n*K*logn)$ which is in $O(n*K*logK*logn)$

g

By 6-(f), Our time complexity is $O(n*K*logn)$.