Q_2

Solutions:

- Assume that the two databases are denoted by A1 and A2. Thus, According to the problem, the function `queries(A,k)` will return the K_{th} smallest value in A.
- A fact about the median value in this question is, the median value m of A1 and A2, always lies in the range of m1 and m2, which are the median values of A1 and A2 separately. That is, if m1 < m2, then, m1 <= m <= m2. This can be easily proved mathematically by the method of "proof by contradiction".

Therefore, we may choose to use the method of "divide and conquer" and solve this problem recursively. That is, in general, for each time of recursion:

- First, find the m1 and m2 respectively.
- Then, through the comparisons of these two values, the range of the m lies can be reduced by half.
- Update the range accordingly, which enables us to keep tracking the median and the portion of the databases that the median m lies in. And since the size of database A1 and A2 are both reduced approximately by half and therefore can be considered as the "new" input of the function. Thus, we can solve it recursively.

Pseudo code:

// assume that the length of database is n and the two database A1 and A2 are global variables, thus we need not to include them in the
// functions. (Of course, it is okay to add them in the input filed of functions)
// the head and tail are the boundaries of each database respectively, head is the smallest and the tail is the largest.

```
Funtion find_medium(int head1,int tail1,int head2,int tail2){        // function name
    m_index1 = head1 + (tail1 - head1)/2        /* index1 and index2 are used to store the index of two databases, and the
    m_index2 = head2 + (tail2 - head2)/2           values are used to store the corresponding value. */
    m_value1 = queries(A1,m_index1)
    m_value2 = queries(A2,m_index2)
    /* we need to find who is larger among m_value1 and m_value2, since we need to judge whether the left half or the right half of
two databases should be kept separately.
    if m_value1 > m_value2
        if (head1 == tail1) && (head2 == tail2) then:    /* the condition that recursion stops and return
            return  m_value2                                the answers. According to the question, the
                                                            smaller one among two possible median value
                                                            will be kept. */
/* next, we need to judge the parity of (tail2 + head2) since when it is odd the database A2(the database with smaller median value in this
   time) will return one more values of possible range. Thus, we need to delete this unnecessary border value to keep the size of A1 and
   A2 equal and keep updating the new boundaries*/
        else
            if (tail2 + head2) % 2 == 0 then:
                return find_medium(head1,m_index1,m_index2,tail2)
            else
                return find_medium(head1,m_index1,m_index2+1,tail2) //add to the head.
    else // when m_value2 > m_value1,the logic mentioned above still applies, just choose the opposite.
        if (head1 == tail1) && (head2 == tail2) then:
            return  m_value1
        else
            if (tail2 + head2) % 2 == 0 then:
                return find_medium(m_index1,tail1,head2,m_index2)
            else
                return find_medium(m_index1+1,tail1,head2,m_index2)
    }
find_medium(1,n,1,n)      // run the code by call this function.
```

Runtime analysis: since this is a divide and conquer method, in each time of recursion, the size will be reduced by half. Therefore, the problem is solved by T(n) = T(n/2) + C, Thus the time used should be $O(\log^n)$.

Q3

Solution:

(a).

According to the example in the course, this problem is similar but with slight different that count the large inversions instead of inversions in an array. Thus, the algorithm will be slightly different, but it can still be solved by merge sort and counting while merges. The different point is, this time, we should count separately, that means, if we only change the if condition from $a_i > a_j$ to $a_i > 2* a_j$, the problem cannot be solved, since it will omit many possible large inversion pairs.

Thus, the logic flow for this "divide and conquer" method is:

- First, we divide the array into two halves, and solve this problem for each two arrays in half of the origin size.
- Second, when it cannot be divided any further, we sort and merge them together, during this process, we will count the number of larger inversions between the two arrays.
- Thus, the total inversions will equals to the number of inversions in each of the arrays, and the "cross inversions" between them

```
Function countInversions(array a)        // the main function
        int[] temp = new int[a.length]        // this is the array used to store the sorted arrays in each merge temperately.
        return mergeSort(a,temp, 0, a.length)        // call function mergeSort, begin the sorting.
/* l_ptr and h_ptr is used to store the pointer which points to the lower boundary and higher boundary of the current array separately. */
Funtion mergeSort (array a, array temp,int l_ptr, int h_ptr)
        if l_ptr == h_ptr - 1 then:     // this means only one elements in this array, no inversions.
           return 0
        mid = (l_ptr + h_ptr)/2     // divide the array into half, store the new boundary.
       return mergeSort (a,temp, l_ptr, mid) + mergeSort (a,temp, mid, h_ptr) + merge (a,temp, low, mid,
h_ptr)        // total inversions equals to the numbers of inversions in each array plus the cross inversions,which are the inversions forms between the arrays.
Function merge (array a,array temp, int l_ptr, int mid, int h_ptr) {
       count = 0
       low_p = l_ptr;              // low_p and the high_p are the pointers used to track the elements in array.
       high_p = mid;
       i = l_ptr;                  // pointer used in temp array.
    /* sorting the array while merge them, since the two input array have already been sorted, thus we can update them simply by comparing their values and copy the smaller one to temp, and move the pointer.
This will be a linear process O(n) */
      while (high_p < h_ptr && low_p < mid){
        if (a[low_p] <= a[high_p])
        {    temp[i++]  = a[low_p];                 // copy the lower half array
             Low_p++;                }
       else {
            temp[i++]  = a[high_p];                 // copy the higher half array
            high_p++;
            }
       }
/* when one of the pointer hit the boundary, we can just update all the remaining elements in the other array to the temp.*/
      While high_p < h_ptr
        temp[i++] = a[high_p++]
      }
      While low_p < mid
        temp[i++] = a[low_p++]
/* reset the pointers */
       low_p = l_ptr;
       high_p = mid;
/* count the larger inversion pairs */
      while(low_p < mid && high_p < h_ptr){
          if(a[low_p] > 2 * a[high_p])
            count = count + (mid - low_p);          / / all the elements between a[mid] and [low_p] shall be counted
            high_p++;
          }
          else{
            low_p++;}
      }
/* copy the temp array back to the original array, make sure the input array to the merge function are already sorted */
      for (int m = l_ptr; m < h_ptr; m++)
            a[m] = temp[m]
       return count
}  // end of the function merge().
```

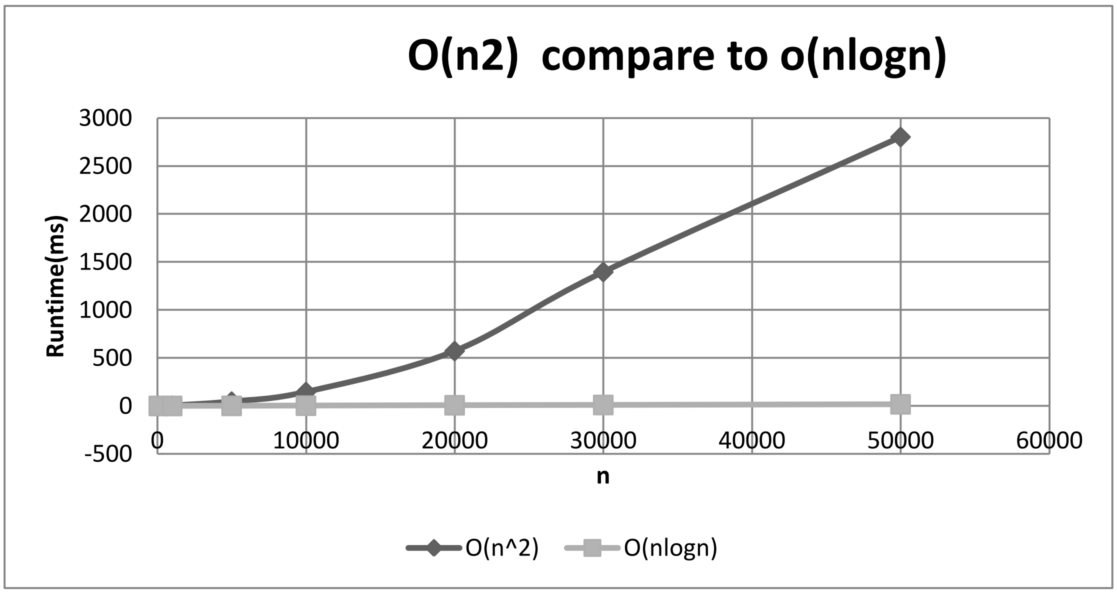
Runtime analysis: this is a "divide and conquer" method. Since the counting of larger inversions and the sorting of the two sorted arrays are all linear time-consumption. Thus, the problem will be solved in a manner of $T(n) = 2*T(n/2) + 2n$, which is $O(n*\log^n)$.

(b).

I code up the program with Java. And the result for this 50000 elements array is 313241073.

(c).

I choose the different length n and count the running time, the graph is shown below:



We can find that, the time difference is very significant when the number of n become larger. And the ratios between the two algorithms obey the n/log ^{n}. Thus It is quite important to find and use time-efficient algorithms when n is large.

Moreover, actually when I implemented this algorithm, at first, I declare the temp array in the merge function, Instead of passing it through the functions. This will not affect the results, but significantly increase the time-consumption, Since it declare much more larger array in total, than n-length one. It will cost around 700ms for 50000 data, which is quite large compare to 17. This teach me that the declaring of large arrays are also quite time-consumption.

Appendix: java code

Question 2:

```
public class question_2
{
      static int[] A1 = {1,2,3,4,5};
      static int[] A2 = {6,7,8,9,10};
public static void main(String[] args)
{
      int length = A1.length;
      int medium = find_medium(1,length,1,length);
      System.out.println(medium);
}
public static int queries(int[] array, int index)
{
      return array[index-1];
}
public static int find_medium(int head1, int tail1,int head2,int tail2)
{
      int m_index1;
      int m_index2;
      m_index1 = head1 + (tail1 - head1)/2;
      m_index2 = head2 + (tail2 - head2)/2;
      int m_value1 = queries(A1,m_index1);
      int m_value2 = queries(A2,m_index2);
      if (m_value1 > m_value2){
            if ((head1 == tail1) && (head2 == tail2))
                  return  m_value2;
            else{
                  if ((tail2 + head2) % 2 == 0)
                        return find_medium(head1,m_index1,m_index2,tail2);
                  else
                        return find_medium(head1,m_index1,m_index2+1,tail2);
            }
      }
      else{
            if ((head1 == tail1) && (head2 == tail2))
                  return  m_value1;
            else{
                  if ((tail2 + head2) % 2 == 0)
                        return find_medium(m_index1,tail1,head2,m_index2);
                  else
                        return find_medium(m_index1+1,tail1,head2,m_index2);
            }
      }
      }
}
```

Question 3:

```
public class CountingInversions {
    public static int countInversions(int[] a) {
        int aa;
        int[] temp = new int[a.length];
         aa =mergeSort(a,temp, 0, a.length);
        return aa;
    }
    private static int mergeSort (int[] a, int[] temp,int l_ptr, int h_ptr) {
        if (l_ptr == h_ptr - 1) return 0;
        int mid = (l_ptr + h_ptr)/2;
        return mergeSort (a,temp, l_ptr, mid) + mergeSort (a,temp, mid, h_ptr) + merge (a,temp, low, mid, h_ptr);
    }
    private static int merge (int[] a,int[] temp, int l_ptr, int mid, int h_ptr) {
      int count = 0;
      int lb = l_ptr;
      int hb = mid;
      int i = l_ptr;
      while (hb < h_ptr && lb < mid){
        if (a[lb] <= a[hb])
        {    temp[i++]  = a[lb];
             lb++;                  }
       else {
            temp[i++]  = a[hb];
            hb++;
            }
       }
      while(hb < h_ptr){
        temp[i++] = a[hb++];
      }
      while(lb < mid)
        { temp[i++] = a[lb++];}
       // System.arraycopy(temp, l_ptr, a, l_ptr, h_ptr - l_ptr);
       lb = l_ptr;
       hb = mid;
      while(lb < mid && hb < h_ptr){
          if(a[lb] > 2 * a[hb]){
            count = count + (mid - lb);
            hb++;
          }
          else{
            lb++;}
      }
      for (int m = l_ptr; m < h_ptr; m++)
       {
            a[m] = temp[m];
       }
       return count;
    }
}
```