Problem 2-1

1. To sort a single list of length k we need 2). We have n/k sublists, so it will take Θ(2)=
2. According to the condition we have n/k sublists of length k which are sorted using insertion sort. Now we will merge it using the standard merging mechanism. So the depth of the merge tree will be log(n)-log(k)= log(n/k) because we will merge sublists which length is at most k . The time for each level of merging will be cn, so algorithm will take Θ(n log(n/k)) time.
3. Given that the modified algorithm runs in Θ (nk+nlog(n/k)) worst-case time. Standard merge sort running-time is Θ(nlogn) or Θ(nlog(n/k)) for this problem according to the part (b) of 2-1. For any constant choice of k, we will have Θ(nk+nlog(n/k))= Θ(n(k+log(n/k))). Since k is constant the latter is the same as Θ(nlog(n/k)).
4. In practice we will get the best k by trying for various values for sufficiently large n and timing it.

Problem 2-2

1. In order to show that bubblesort actually sorts, we also need to prove that A’ contains the same elements as A. The only modification that we make to array is swapping it’s elements, so in A’ we will have the same elements but in a different arrangement. So it’s true that A’ contains the same elements as A.
2. BUBBLESORT(A)  
   1.for i=1 to A:length-1   
   2. for j=A.length down to i+1  
   3. If A[j]<A[j-1]  
   4. Exchange A[j] with A[j-1]  
   We need show that the loop in lines 2-4   
   *Initialization:* The loop invariant holds before the first loop iteration, when j=A.length. The smallest element is in A[1…A.length], the smallest element is before j so the loop invariant holds prior to the first iteration of the loop.  
   *Maintenance:* We need to show that each iteration maintains the loop invariant.So that is what’s going on in one step of the loop: if the element is not in it’s place it will be moved in one place closer to it sorted place.We find the smallest element A[i+1…A.length] and move it to it’s sorted place. And of course the values before ith element will be smaller than ith element when the loop ends, so the loop invariant holds prior.  
   *Termination:* The loop will terminate if j<i+1. As we say, the values before ith element will be smaller than ith element when the loop ends. So the ith element will be in it’s place and that is why we execute that loop. After ending the first **for** loop we will have sorted array.
3. *Initialization*: The loop invariant holds before the first loop iteration, when i=1. At the start of each iteration he subarray A[1…i-1] consists of i-1 smallest elements of A in sorted order. If i=1 the first 0 elements are sorted trivially, which shows that the loop invariant holds prior to the first iteration of the loop.  
   *Maintenance*: We need to show that each iteration maintains the loop invariant. The body of the first **for** loop works by moving first smallest element to the beginning of the array, the second smallest to the second place and so son. So the body of the first for loop works by moving each element in its necessary place. After each iteration the subarray A[1..i] consists of the i smallest elements of the array in sorted order. Incrementing i for the next iteration of the **for** loop then preserves the loop invariant. About the second loop we have spoken in part (b).  
   *Termination*: The loop will terminate if i>A.length-1=n-1 (A.length=n). Because each loop iteration increases I by 1, we must have j=n at that time. Substituting n for i in the wording of loop invariant, we have that the subarray A[1..n-1] consists of the n-1 smallest elements of the array but in sorted order, so the last one A[n] will be the biggest element, so our array will be sorted after termination. Hence, the algorithm is correct.  
   As I show that the algorithm is correct , after termination we will have our array sorted in ascending order, so
4. Worst-case running time of bubblesort is Θ(n2) because we have n-2 iterations for the second loop if i=1. Best-case running time of bubblesort is Θ(n2), whereas best-case running time of insertion sort is Θ(n).

Problem 2-3

1. Given the coefficients a0,a1,…,an and a value for x. The Horner’s rule is  
   1. y=0  
   2. for i=n down to 0  
   3. y=ai+x\*y (\* is multiple)  
   So it has runtime Θ(n).
2. Naive polynomial evaluation algorithm P(x)=a0+a1x+a2x2+…+anxnPseudo-code will be   
   1. y=0  
   2. for i=1 to A.length  
   3. p=1  
   4. for j=1 to i  
   5. p=p\*x   
   6. y=y+ai\*p  
   This pseudo-code has runtime Θ(n2) because it will compute the powers of x, if i=A.length=n, **j** will go from 1 to n. Clearly that is harder than Horner’s rule.
3. We have k+i+1xk at the start of each iteration. As we see in part (a) of 2-3 i=n, k+n+1xk= k+n+1xk=an+1+an+2x1, since *a*n+1 = *a*n+2=0, after the first iteration when i=n, y=0. Suppose it is true for i, after ith iteration we will have   
   y=*a*i+k+i+1xk = *ai*+xk+ixk=k+ixk . The loop will be terminated if i=0, so the final result will be y=kxk .
4. In the part (c) of 2-3 we get y=kxk =a0+a1x+a2x2+…+anxn which evaluates a polynomial characterized by the coefficients a0,a1,…,an .

Problem 2-4

1. Inversions are (2, 1),(6,1), (3, 1), (8, 6), (8, 1).
2. The array {n,n-1,…,1} has the most inversions. The nth element has inversion with the rest n-1 elements, the (n-1)th element has inversion with the rest n-2 elements and so on. So the count of all possible inversion will be n-1+n-2+…+1= .
3. The second loop of the insertion sort algorithm will execute once for the elements for which the condition of the inversion is satisfied: if (i<j) and A[i]>A[j]. The count of all inversions is the count of all swaps in insertion algorithm.
4. I modified the merge sort. Here is the pseudo-code and after that the code in java  
   ***merge(arr,p,q,r)***
5. c=0
6. n1=q-p+1
7. n2=r-q
8. let L[1..n1+1] and R[1..n2+1] be new arrays
9. for i=1 to n1
10. L[i]=arr[p+i-1]
11. for i=1 to n2
12. R[i]=arr[q+i]
13. i=1
14. j=1
15. k=p
16. while i<n1 and j<n2
17. if(L[i]≤R[j])
18. arr[k]=L[i]
19. i=i+1
20. else
21. arr[k]=R[j]
22. j=j+1
23. c=c+j //number of inversions between left and right arrays
24. k++
25. while i<n1
26. arr[k]=L[i]
27. k=k+1
28. i=i+1
29. c=c+n2 //if we reached to this cycle and the condition is true that means that i<n1 and j>n2
30. //that means that each element from the left array has inversion with the whole right
31. // array and the size of the right array is n2
32. while j<n2
33. arr[k]=R[j]
34. k=k+1
35. j=j+1
36. return c

***merge\_sort(arr,p,r)***

1. count=0 //number of all inversions
2. if p<r
3. q=p+(r-p)/2
4. merge\_sort(arr,p,q)
5. merge\_sort(arr,q+1,r)
6. count=count+merge(arr,p,q,r)
7. return count

class MergeSort {  
  
 int merge(int[] arr**,** int p**,** int q**,** int r)  
 {  
 int c=**0;** // number of inversions for halves that we will merge  
  
 int n1 = q - p + **1;** int n2 = r - q**;** int[] L = new int[n1]**;** int[] R = new int[n2]**;** System.*arraycopy*(arr**,** p**,** L**, 0,** n1)**;** System.*arraycopy*(arr**,** q+**1,** R**, 0,** n2)**;** int i = **0,** j = **0;** int k = p**;** while (i < n1 && j < n2) {  
 if (L[i] <= R[j]) {  
 arr[k] = L[i]**;** i++**;** }  
 else {  
 arr[k] = R[j]**;** j++**;** c+=j**;** //number of inversions between left and right arrays  
 }  
 k++**;** }  
  
 while (i < n1) {  
 arr[k++] = L[i++]**;** c+=n2**;**// if i<n1 and j>n2 that means that the each element from the rest elements  
 // in left array has inversions with the whole right array. The size of the right array is n2  
 }  
  
 while (j < n2) {  
 arr[k++] = R[j++]**;** }  
 return c**;** }  
  
  
 int merge\_sort(int[] arr**,** int p**,** int r)  
 {  
 int count=**0;** //number of all inversions  
 if (p < r) {  
 int q = p + (r - p) / **2;** merge\_sort(arr**,** p**,** q)**;** merge\_sort(arr**,** q + **1,** r)**;** count+=merge(arr**,** p**,** q**,** r)**;** }  
 return count**;** }  
  
  
  
 public static void main(String[] args)  
 {  
 int[] arr = { **2, 3, 8, 6, 1** }**;** MergeSort ob = new MergeSort()**;** int count=ob.merge\_sort(arr**, 0,** arr.length - **1**)**;** for (int j : arr) System.*out*.print(j + " ")**;** System.*out*.println()**;** System.*out*.println(count)**;** }  
}