Part 1:

3.For the selection sort:

```
public int selectionSort( )                                           Statement
 {

   int counter = 0;                                                     (1)
   int temp, indexOfMax;                                                (2)

   for ( int i = 0; i < arr.length - 1; i++ )                           (3)
   {
     indexOfMax = 0;                                                    (4)

     for ( int j = 1; j < arr.length - i; j++ )                         (5)
     {
                if ( arr[j] > arr[indexOfMax] )                         (6)
                       indexOfMax = j;                                  (7)
                counter++;                                              (8)

     }
          temp = arr[indexOfMax];                                       (9)
          arr[indexOfMax] = arr[arr.length - i - 1];                    (10)
          arr[arr.length - i - 1] = temp;                               (11)
   }
   return counter;                                                      (12)

 }
```

Statement	#Time Executed	Operations
(1)	1	Assignment
(2)	0	
(3)	1+(n+1)+1+n = 2n+3	Assignment, comparison and array length, Math "-", variable increment
(4)	1	Assignment
(5)	n(1+n+1+1+n) = 2n^2 + 3n	Assignment, comparison and array length, Math "-", variable increment
(6)	3n*n	2 x Access array, comparison
(7)	0 to n*n	Assignment
(8)	n*n	Increment
(9)	2n	Assignment, access array
(10)	6n	Assignment,2 x access array, array length,2 x math "-"
(11)	5n	2 x Math "-", Array access, assignment, array length
(12)	1	Return

Worst case: 7n^2 + 18n + 6
Best case: 6n^2 + 18n + 6
Average case: 6.5n^2 + 18n +6

Its running time is ***O(n^2)***

Part 2:

2.1 Algorithm A executes 10nlogn operations, while algorithm B executes n^2 operations. Determine the minimum integer value n0 such that A executes fewer operations than B for n ≥ n0.

Firstly, 10nlogn < n^2, which is equal to 10logn < n, if we divide both sides by n. So we can just compare 10logn and n.
Secondly, because log is related to the power of 2 (log8 = 3, 2^3 =8 for example), it will be easier to compare the runtimes using numbers which are the result of powers of 2. The table below shows where n finally becomes larger than 10logn.

n	n	10logn
2^0	1	0
2^1	2	10
2^2	4	20
2^3	8	30
2^4	16	40
2^5	32	50
2^6	64	60

From this table, we can see that when n <=32, 10logn > n, and when n = 64, 10logn < n, so I can see that the minimum number "n" where 10logn executes fewer operations than n is between 50 and 60.

Then I used log base 2 calculator:
n= 55:
10 log55 = 57.81 > 55
n= 57:
10 log57 = 58.32 > 58
n= 58:
10 log58 = 58.58 > 58
n= 59:
10 log59 = 58.83 < 59

In the range of 50 to 60, I assigned different integer value to n. Finally, I got the minimum integer value is ***59*** that A executes fewer operations than B.

2.2 Comment and justify the running time of the following algorithm:

Algorithm Foo (a, n):

```
Input: two integers, a and n
Output: ?
k←0                 (1)
b←1                 (2)
while k<n do        (3)
    k←k + 1         (4)
    b←b * a         (5)
return b            (6)
```

The output is a^n

Statement	Time executed	Operations
(1)	1	Assignment
(2)	1	Assignment
(3)	n	Iterate the while loop
(4)	0 to 2n	Math "+", assignment
(5)	0 to 2n	Math "-", assignment
(6)	1	return

Best case: 3+5n. (when k = 0, it didn't have any operations after the while loop)
Worst case: 3+3n
Average case: 3+4n

Its running time is ***O(n)***

2.3 Comment and justify the running time of the following algorithm:

```
Algorithm Bar (a, n):
   Input: two integers, a and n
   Output: ?
   k←n
   b←1
   c←a
   while k > 0 do
        if k mod 2 = 0 then
                k←k/2
                c←c * c
        else
                k←k – 1
                b←b*c
        return b
```

The output is a^n

Situation 1:
If n is even, it get halved, else it minus 1 and get halved in the next iteration.
A number can only divided in half log n times (e.g. if n = 8, log 8 = 3, it can be halved 3 times, so the running time is logn), when n is even, the situation of "else" part's operation is constant, it is always 1, because the decrement only operates when k = 1.

Situation 2:
If a number n is odd, the divide part's running time is still logn, the decrement part's running time is constant, it is always 2, because the decrement only operates when k equals the odd number and 1 (e.g. if n = 9, 9-1 =8, is an operation and then iterate the division process when n = 8). Then decrement when k=1.

Situation 3:
When a number needs to be decremented in between halving it, (e.g. n = 6. 6/2 is 3, so it will still do decrement once, and do another time when k = 1), the longest running time is still logn + 2 .

Its running time is ***O(logn)***