1. (Question 2-2 from the text book)

a. We still need to show that A' is a permutation of all elements of A.

b.

Loop invariant: At the beginning of each iteration of the **for** loop of lines 2-4, A[j] = min {A[k] : j ≤ k ≤ n}, and A[j... n] is a permutation of the original elements in A[j... n] at the time when the loop started.

Initialization: Prior to the first iteration of the loop, j = n, so that the subarray A[j... n] consists of single element A[*n*], which shows the loop invariant holds.

Maintenance: Consider an iteration for a given value of j. A[*j*] is the smallest value in A[j... n]. In lines 3-4, if A[j] is less than A[j-1], exchange A[j] and A[j-1], and so A[j-1] will be the smallest element in A[j-1... n] afterward. Since it is the only possible change to the subarray A[j-1... n], and the subarray A[j... n] is a permutation of the elements that were in A[j... n] at the time when the loop started, now that A[j-1... n] is a permutation of the elements that were in A[j-1…n] at the time that the loop started. Decrementing j for the next iteration maintains the invariant.

Termination: At termination, j=i. A[i] = min{A[k]: i ≤ k ≤ n} and A[i... n] is a permutation of the elements that were in A[i... n] at the time when the loop started.

c.

Loop invariant: At the beginning of each iteration of the for loop of lines 1-4,the subarray A[1... i-1] contains the i-1 smallest elements originally in A[1... n], in sorted order, and A[i... n] contains the n-i+1 remaining elements originally in A[1... n].

Initialization: Prior to the first iteration of the loop, i = 1. The subarray A[1... i-1] is empty, and so that the loop invariant holds.

Maintenance: Consider an iteration for a given value of i. By the loop invariant, A[1... i-1] contains the i smallest elements in A[1... n], in sorted order. Part(b) showed that after executing the for loop of lines 2-4, A[i] is the smallest element in A[i... n], and so A[1... i] is now the i smallest values originally in A[1... n], in sorted order. In addition, since the for loop of lines 2-4 permutes A[i... n], the subarray A[i+1... n] consists of the n-i remaining elements originally in A[1... n]. Incrementing i for the next iteration maintains the invariant.

Termination: At termination, i = n+1, so that i-1= n. A[1... i-1] is the entire array A[1... n], and it contains the original array A[1... n], in sorted order.

d. The basic operation is the exchange operation. For a given value of i, the number of iterations of the for loop of lines 2-4 is n-i. Thus, the total number of exchange operation is

$$\sum_{i=1}^{n}(n-i)=\sum_{i=1}^{n}n-\sum_{i=1}^{n}i=n^2-\frac{n(n+1)}{2}=\frac{n^2}{2}-\frac{n}{2}$$

Therefore, the worst-case running time of bubblesort is $\Theta(n^2)$, which is the same as that of the insertion sort. □

2. (Question 3-2 from the text book):

A	B	O	o	Θ	ω	Ω
$(\lg n)^k$	n^k	Yes	Yes	No	No	No
n^k	c^n	Yes	Yes	No	No	No
$n^{0.5}$	$n^{\sin n}$	No	No	No	No	No
2^n	$2^{n/2}$	No	No	No	Yes	Yes
$n^{\lg c}$	$c^{\lg n}$	Yes	No	Yes	No	Yes
ln (n!)	$\lg (n^n)$	Yes	No	Yes	No	Yes

(Question 3-4 from the text book): Prove or disprove following conjectures:

a.) f(n) = O (g(n)) implies g(n)= O(f(n)) **FALSE**

Let f(n) = n and g(n) = n^2

In this case, n = O (n^2) but $n^2 \neq$ O (n)

Hence, false.

b.) f(n) + g(n) = Θ(min (f(n), g(n))) **FALSE**

Let f(n) = n^2 and g(n)= n^3

f(n) + g(n) = $\mathbf{n^2 + n^3}$

Also, Θ min (f(n), g(n)) = Θ (min (n^2, n^3)) = $\mathbf{\Theta(n^2)}$

Clearly, f(n) + g(n) ≠ Θ(min (f(n), g(n))) c.) $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n))$ **TRUE**

Given that $f(n) = O(g(n))$ implies that for some $c>0$, $n \geq n_0 \geq 0$.

We have

$0 \leq f(n) \leq c\, g(n)$

Taking log both sides:

$\lg (f(n)) \leq \lg (c\, g(n))$

⇨ $\lg (f(n)) \leq \lg c + \lg (g(n))$

Since $\lg(g(n)) \geq 1$, therefore, $\lg c \,.\, \lg (g(n)) \geq \lg c$

Next, $\lg ((f(n)) \leq \lg (g(n)) + \lg c \,.\, \lg (g(n))$

⇨ $\lg ((f(n)) \leq (1+ \lg c) \lg(g(n))$

⇨ $\lg ((f(n)) = O(\lg (g(n)))$

Hence, true.

d.) $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$ **FALSE**

Let $f(n) = 2n$ and $g(n) = n$

Clearly, $f(n) = O(g(n))$

But $2^{2n} \neq O(2^n)$

Hence, false.

e.) $f(n) = O((f(n))^2)$ **FALSE**

Let us assume that $f(n) = O((f(n))^2)$ is true. Therfore,

$f(n) \leq c((f(n))^2$

$(1/f(n)) \leq c$ for all $n \geq n_0$

Such c would exist only if 1/min (f(n))is finite for all values of n, even when n is infinitely large or zero. Otherwise, not.

Therefore, this holds true only if the function has a non-zero minimum value for all values $n \geq n_0$

Hence, false.

f.) $f(n) = O(g(n))$ implies $g(n) = \Omega (f(n))$ **TRUE**

By definition, if $f(n) = O(g(n))$ then,

$\mathbf{0} \leq \mathbf{f(n)} \leq c\, g(n)\,)$

=> $\mathbf{0} \leq (1/c)\, \mathbf{f(n)} \leq g(n)\,)$

=> $g(n) \geq (1/c)\, \mathbf{f(n)} \geq 0$

=> $g(n) = \Omega(f(n))$

Hence, true.

g.) f(n) = Θ(f(n/2)) **FALSE**

Let f(n)= 2^{2n}

Therefore, f(n/2) = 2^{n}

Clearly, $\mathbf{2^{2n}} \neq \Theta(\mathbf{2^{n}})$

Hence, false

3.

a) Functions in increasing order of asymptotic growth:

2^{-3n},

$n^{-n} + \log(n) + 2$,

$\log(n) + 1$,

$\log(n) + \sqrt{n} + 12$,

$2n - 10$,

n^2,

$n - n^3 + 7n^5$,

$n^{n/5}$,

$n!$

b) Out of the above functions, $\boldsymbol{n^{-n} + \log(n) + 2}$ and $\boldsymbol{\log(n) + 1}$ are exchangeable since they are in the same asymptotic class; i.e. $\boldsymbol{n^{-n} + \log(n) + 2}$ is $\boldsymbol{\Theta(\log(n) + 1)}$ and vice versa.

4.

Hint: Apply l'Hôpital's rule "t" times.

This reduces the term n^t to $t!$

Therefore,

$$\lim_{n \to \propto} \frac{e^n}{n^t}$$

$$= \lim_{n \to \propto} \frac{e^n}{t!} = \propto$$

5. Algorithm to find a^n in logn time:

```
Fast_Power(integer a, n)
{
        Integer i;
        Integer Product;

        //Assume 'n' is a power of 2
        // log2(n) returns the logarithm to the base 2 of 'n'

        Product = a;
        For i=1 to log2(n) do
                Product = Product * Product;
        End For

        return Product;
}
```

This algorithm calculates a^n in logn executions of the loop.

The basic operation is multiplication

Multiplication times: log n

So the running time is $\Theta(\log n)$

<u>Alternate Solution:</u>

Pseudocode

```
product ← a;
i ← 1
while i<n
        product ← product × product
        i ← i × 2
Endwhile
return(product)
```

Loop invariant: At the beginning of each iteration, $product = a^i$ for $1 \le i \le n$

Initialization: Prior to the first iteration of the loop, i=1 and product=a, which shows the loop invariant holds.

Maintenance: Consider an iteration for a given value of i=k. Before this iteration, $product = a^k$. Then in line 4, product becomes a^{2k} and i changes to 2k. So before the iteration for i=2k, $product = a^{2k}$. Multiply i by 2 for the next iteration maintains the invariant.

Termination: At termination, i=n, so that $product = a^n$

Time complexity:

The basic operation is multiplication

Multiplication times: log n

So the running time is $\Theta(\log n)$

6.

a) Assuming that the `merge` procedure takes $(i + j - 1)$ time to merge two sorted arrays of size i and j, we can list the times taken at each step by the given algorithm as follows:

Step 1	$2n - 1$
Step 2	$3n - 1$
Step 3	$4n - 1$
...	$\dots$
Step k-1	$kn - 1$
Total	$2n + 3n + \cdots + kn - (k - 1)$

We can simplify the total as follows:

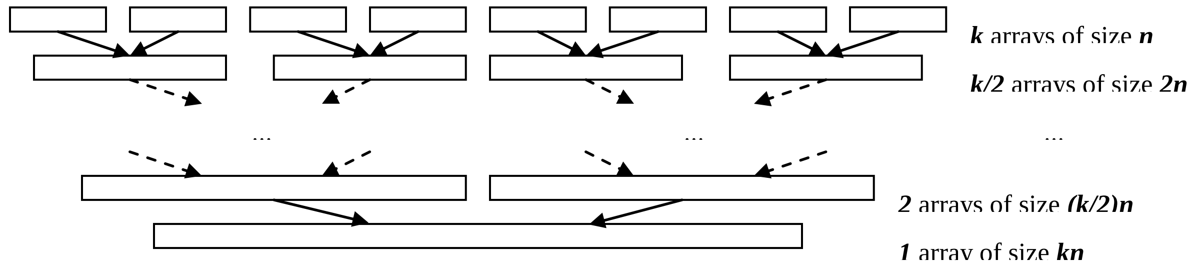
$T(n, k) = -n + n + 2n + 3n + \cdots + kn - (k - 1)$

$= n(1 + 2 + 3 + \cdots + k) - (n + k - 1)$

$$= n \cdot \frac{k(k + 1)}{2} - (n + k - 1)$$

Therefore, $T(n, k)$ is $O(nk^2)$.

b) A more efficient approach would be to simply merge the arrays pair-wise, as illustrated below:



Listing the times in each step, we have: (assume k is a power of 2)

Step 1	$(k/2)\cdot(2n-1) = kn - k/2$
Step 2	$(k/4)\cdot(4n-1) = kn - k/4$
Step 3	$(k/8)\cdot(8n-1) = kn - k/8$
...	$\dots$
Step $\log_2 k$	$1\cdot(kn-1) = kn - 1$
Total	$kn\cdot\log_2 k - (k/2 + k/4 + \cdots + 1)$

Therefore,

$$T'(n,k) = kn\cdot\log_2 k - \sum_{i=1}^{\log_2 k} k\cdot(1/2)^i$$

$$= kn\cdot\log_2 k - \frac{(k/2)\cdot\left[1-(1/2)^{\log_2 k}\right]}{1-(1/2)}$$

$$= kn\cdot\log_2 k - k\cdot[1-(1/k)]$$

$$= kn\cdot\log_2 k - k + 1$$

which is $O(nk\log_2 k)$. This is clearly better than $O(nk^2)$.

A recursive algorithm for this would look like:

MultiMerge(**L** : array of lists to merge, k : integer indicating how many lists there are in **L**)

    If k == 1   // i.e. there is only one list
        Return **L**[0]
    Else
        L1 = MultiMerge(**L**[0 : k/2], k/2)   // Do a "multi-merge" of the first k/2 lists
        L2 = MultiMerge(**L**[(k/2)+1 : k], k/2)   // Ditto for the last k/2
        Return MERGE(L1, L2)   // merge the results (i.e., merge two lists of size k)

The time complexity of this algorithm can be computed using:

$T(n,k) = 2T(n,k/2) + kn - 1$

Note that the algorithm's time complexity really only depends on k, so we can write it as $n \cdot T^{'}(k)$ where $T^{'}(k) \leq 2T^{'}(k/2) + k$

And this is clearly $O(k \log_2 k)$.