Problem 1
[15 pts] Solve the following recurrence relations. You do not need to give a $\Theta()$ bound for (a) and (b); it suffices to give the $O()$ bound that results from applying the Master theorem. You may assume that $T(n) = O(1)$ for $n = O(1)$.

1. $T(n) = 2T(n/3) + 1$
2. $T(n) = 7T(n/7) + n$
3. $T(n) = T(n-1) + 2$

Solution:

Theorem 1 (The mater theorem)**.** *Let* $a \geq 1$, $b > 1$, $f(n) = O(n^d)$ *where* $d \geq 0$, *and* $c = log_b a$

$$T(n) = \begin{cases} O(1) & \textit{if } n = O(1) \\ aT(n/b) + f(n) & \textit{otherwise} \end{cases} \tag{1}$$

1. $c < d$: $T(n) = O(f(n)) = O(n^d)$

2. $c > d$: $T(n) = O(n^c)$

3. $c = d$: $T(n) = O(n^c lgn)$

1. Applying the Master Theorem with $a = 2$, $b = 3$, $c = log_3 2 \approx 1.584962501$ and $d = 0$. We get $c > d$, therefore $T(n) = O(n^c) = O(n^{log_3 2})$
2. Applying the Master Theorem with $a = 7$, $b = 7$, $c = log_7 7 = 1$ and $d = 1$. We get $c = d$, therefore $T(n) = O(n^c lgn) = O(nlgn)$.
3. We can not apply the Master Theorem on this part because $b \leq 1$, hence we use Iteration/Recursion-Tree method instead.
   According to the starting point
   $$T(n) = T(n-1) + 2 \tag{2}$$
   we derive
   $$T(n-1) = T(n-2) + 2 \tag{3}$$
   We substitute $T(n-1)$ in equation (2) with $T(n-2) + 2$ to get
   $$T(n) = (T(n-2) + 2) + 2 = T(n-2) + 4 \tag{4}$$
   Suppose we iterate this process with k times, we will get
   $$T(n) = T(n-k) + 2k \tag{5}$$
   Lastly we stop iterating when k equal to the problem size at the base case, so that $n = k$ and $T(k) = T(0) + 2k = 1 + 2k = O(2k) \approx O(n)$.

Problem 2
[15 pts] Give a recursive version of the algorithm Insertion-Sort based on the following paradigm: to sort $A[1..n]$, we first sort $A[1..n-1]$ recursively and then insert $A[n]$ in its appropriate position. Write a pseudocode for the recursive version of Insertion-Sort and analyze its running time by giving a recurrence relation for the running time and then solving it.

Solution:

```
def insertionSort(A[1..n], n):
    if $n \leq 1$ then
        return;
    end
    insertionSort(A[1..n], n - 1)
    lastInteger $\leftarrow A[n-1]$
    i $\leftarrow n-2$
    while $i \geq 0$ and $A[i] > lastInteger$ do
        A[i + 1] = A[i];
        i $\leftarrow$ i - 1;
    end
    A[i + 1] $\leftarrow$ lastInteger
```

There are two steps in this recursive sorting algorithm:

1. Sort the sub-array $A[1..n-1]$.
2. Insert $A[n]$ into the sorted sub-array from step 1 in proper position.

For $n = 1$, step 1 doesn't take any time as the sub-array is an empty array and step 2 takes constant time, i.e. the algorithm runs in $\Theta(1)$ time.
For $n > 1$, step 1 again calls for the recursion for $n-1$ and step 2 runs in $\Theta(n)$ time.
Thus, we can write the recurrence as:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1) \\ T(n-1) + n & \text{if } n > 1 \end{cases} \tag{6}$$

So for any general $n > 1$,

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= (T(n-2) + (n-1)) + n \\ &= T(1) + (n + (n-1) + (n-2) + (n-3) + ... + (n-n)) \\ &= 1 + \sum_{k=0}^{n} k \\ &= 1 + \frac{n(n-1)}{2} \\ &= O(n^2) \end{aligned} \tag{7}$$

Problem 3
[15 pts] Let A be an array of n numbers. Use the algorithm Select(A, k) for finding the k-th smallest element of any array A of numbers to modify Quick Sort so that it runs in $O(nlgn)$ time in the worst case. Write a pseudocode for the modified Quick Sort, and analyze its running time by describing its recurrence relation and solving it. You can use the algorithm/subroutine Select() as a black box in the modified Quick Sort.

Solution:

```
def QuickSort(A[1..n], low, high):
    if $low < high$ then
        location ← Partition(A[1..n], low, high)
        QuickSort(A[1..n], low, location)
        QuickSort(A[1..n], location, high)
    end
```

```
def Partition(A[1..n], low, high):
    pivot = Select[A[low, high], (high - low)/2]
    leftwall = low for $i = low + 1$ to high do
        if A[i] $<$ pivot then
            swap(A[i], A)
            leftwall ← leftwall + 1
        end
        swap(pivot, A[leftwall])
    end
    return leftwall
```

There are two steps in this recursive sorting algorithm:

1. Sort the sub-array base on the pivot location (on both left and right side)
2. Finding the location of pivot through partition function

For $n = 1$, step 1 doesn't take any time as the sub-array is an empty array and step 2 takes constant time, i.e. the algorithm runs in $\Theta(1)$ time.
For $n > 1$, step 1 again calls for the recursion for $n/2$ two times and step 2 takes $\theta(n)$ time to compare the whole array.
Thus, we can write the recurrence as:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1) \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases} \tag{8}$$

By using Select() method, we are guaranteed to have median number of array for efficient sorting, then by applying the Master Theorem with $a = 2$, $b = 2$, $c = log_2 2 = 1$ and $d = 1$. We get $c = d$, therefore $T(n) = O(n^c lgn) = O(nlgn)$.

Problem 4

[15 pts] Give an algorithm that takes as input a positive integer n and a number x, and computes x^n (i.e., x raised to the power n) by performing $O(lgn)$ multiplications. Your algorithm CANNOT use the exponentiation operation, and may use only the basic arithmetic operations (addition, subtraction, multiplication, division, modulo). Moreover, the total number of basic arithmetic operations used should be $O(lgn)$.

Solution: In order to get $O(logn)$ time, we have to to divided the whole into sub-problems. Solving x^n is equivalent to solve $x^{n/2} \cdot x^{n/2}$ and $x^{n/4} \cdot x^{n/4} \cdot x^{n/4} \cdot x^{n/4}$ and so on until $n = 0$.

```
def Power(x, n):
    if $n == 0$ then
        return 1
    end
    temp ← Power(x, n/2)
    if $n$ is odd then
        return $x$ * temp * temp
    end
    if $n$ is even then
        return temp * temp
    end
```

Problem 5

[**20 pts**] Although merge sort runs in $\Theta(nlgn)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

1. [**5 pts**] Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

2. [**10 pts**] Show how to merge the sublistsin $\Theta(nlg(n/k))$ worst-case time.

3. [**5 pts**] Given that the modified algorithm runs in $\Theta(nk + nlg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

Solution:

1. For input of size k, insertion sort runs on $\Theta(k^2)$ worst-case time. So, worst-case time to sort n/k sublists, each of length k, will be $n/k \cdot \Theta(k^2) = \Theta(nk)$.

2. We have n elements divided into n/k sorted sublists each of length k. To merge these n/k sorted sublists to get a single sorted list of length n, we have to take 2 sublists at a time and continue to merge them. This will result in $\lg(n/k)$ steps and in every step, we are essentially going to compare n elements. So the whole process will run at $\Theta(n \lg(n/k))$.

3. For the modified algorithm to have the same asymptotic running time as standard merge sort. To satisfy this condition, k cannot grow faster than $\lg nlgn$ asymptotically, if it does then because of the nk term, the algorithm will run at worse asymptotic time than $\Theta(n \lg n)$.
   Let's assume, $k = \Theta(\lg n)$,

$$\begin{aligned}\Theta(nk + n\lg(n/k) &= \Theta(nk + n\lg(n) - n\lg(k)) \\ &= \Theta(n\lg(n) + n\lg(n) - n\lg(\lg(n))) \\ &= \Theta(2n\lg(n) + n\lg(\lg(n)) \\ &= \Theta(n\lg(n))\end{aligned} \tag{9}$$

   $\lg(\lg(n))$ is very small compared to $\lg(n)$ for sufficiently larger values of n.

Problem 6
[20 pts] A group of n Ghostbusters is battling n ghosts. Each Ghostbuster carries a proton pack, which shoots a stream at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits the ghost. The Ghostbusters decide upon the following strategy. They will pair off with the ghosts, forming n Ghostbuster-ghost pairs, and then simultaneously each Ghostbuster will shoot a stream at his chosen ghost. As we all know, it is very dangerous to let streams cross, and so the Ghostbusters must choose pairings for which no streams will cross. Assume that the position of each Ghostbuster and each ghost is a fixed point in the plane and that no three positions are collinear.

1. **[10 pts]** Argue that there exists a line passing through one Ghostbuster and one ghost such that the number of Ghostbusters on one side of the line equals the number of ghosts on the same side. Describe how to find such a line in $O(n \lg(n))$ time.
2. **[10 pts]** Give an $O(n^2 \lg(n))$-time algorithm to pair Ghost- busters with ghosts in such a way that no streams cross.

Solution:

1. Find the bottom, left-most point as in Graham scan. Sort the remaining points (by angle) from that point. Assume that the bottom, left most point is a Ghostbuster. Visit the sorted points by increasing angle, keeping track of the difference between number of visited Ghostbusters and Ghosts. Stop when the difference is -1, and connect the point to the bottom, left-most point. Run time is dominated by the sort, which takes $O(n \lg n)$-time.
2. Using the above algorithm matches one pair of Ghostbuster and Ghost. On each side of the line formed by the pairing, the number of Ghostbusters and Ghosts are the same, so use the algorithm recursively on each side of the line to find pairings. The worst case is when, after each iteration, one side of the line contains no Ghostbusters or Ghosts. Then, we need $n/2$ total iterations to find pairings, giving us an $P(n^2 \lg n)$-time algorithm.