Problem # 1

Problem Description: Return the median value of two salaries across two towns.

Inputs:

- n = number of residents in each town.
- $t1$ = database containing salaries for residents of town one.
- $t2$ = database containing salaries for residents of town two.

Outputs: Median salary over both towns.
Assumptions:

- Each town has the same number of people.
- Each salary is unique.
- The median is the n^{th} highest salary across both towns.

Strategy Overview: I will use a divide and conquer angorithm that repeatedly divides the problem into smaller subproblems.

Algorithm Description: In the code below a query for the kth lowest salary in sadtown is made by sad[k]. Similar for querying happytown.

```
def bruteForce(sad, happy, sadx, happyx):
       return max(sad[sadx+1], happy[happyx+1])

def findMedian(sad, happy, sadStart, sadEnd, happyStart, happyEnd):
    if (sadEnd - sadStart == 0):
        return bruteForce(sad, happy, sadEnd, happyEnd)

    if (sadEnd - sadStart)%2 == 0:
        sadMed = sad[int((sadEnd + sadStart)/2)+1]
        happyMed = happy[int((happyEnd + happyStart)/2)+1]

        if sadMed > happyMed:
            sadEnd = int((sadEnd + sadStart)/2)
            happyStart = int((happyEnd + happyStart)/2)

        else:
            sadStart = int((sadEnd + sadStart)/2)
            happyEnd = int((happyEnd + happyStart)/2)

    else:
        sadMed = sad[int((sadEnd + sadStart)/2)+2]
        happyMed = happy[int((happyEnd + happyStart)/2)+2]

        if sadMed > happyMed:
```

```
            sadEnd = int((sadEnd + sadStart)/2)
            happyStart = int((happyEnd + happyStart)/2)

        else:
            sadStart = int((sadEnd + sadStart)/2)+1
            happyEnd = int((happyEnd + happyStart)/2)

    return findMedian(sad, happy, sadStart, sadEnd, happyStart, happyEnd)


sadStart = happyStart = 0
sadEnd = happyEnd = n

print findMedian(sad, happy, 0, len(sad)-1, 0, len(happy)-1)
```

Problem # 2

$$T(n) = T(n/2) + O(1)$$

$k = 0$ so it's case two of the master theorem and the complexity class is $O(\log n)$.

Problem # 3

Correctness will be proven by induction on n.
Base Case: $n = 1$. In this case, when each town has one member, the algorithm simply returns the maximum of the two salaries. This is the median because it is larger than one salary.
Inductive Hypothesis: Assume that the algorithm correctly finds the median for $i < k$ for some $k \geq 1$.

Inductive Step: Consider a situation where $n = k$. We can find the median of each town by simply querying the $\lfloor n/2 \rfloor + 1$ salary from that town. Call the median salary from sadtown s_s and the median salary from happytown s_h. We have to consider the case when n is even and the case when n is odd. First consider when n is odd. Assume without loss of generality that s_s is less than s_h. We can conclude that the overall median is at least as large as s_s and at least as small as s_h. By definition of the median there are $(n+1)/2$ salaries in sadtown that are higher than s_s. Because $s_h > s_s$ There are at least $(n+1)/2$ salaries in happy town that are larger than s_s. Thus in total there are a minimum of n salaries that are higher than s_s and s_s is the smallest possible minimum.

Similarly, by definition of the median there are $(n+1)/2$ salaries in happytown that are smaller than s_h. Because $s_s < s_h$ there are at least $(n+1)/2$ salaries in sadtown that are less than s_h. Thus in total there are at least n salaries that are less than s_h and s_h is the highest possible median.

This means that we can consider the subproblem which ignores the $(n-1)/2$ salaries less than s_s and the $(n-1)/2$ salaries greater than s_h (this is done with pointers in this case) and the resulting set of salaries will still contain the true median. Furthermore, because we have discarded the same number of salaries below and above the true median the resulting set will have the same median. By the inductive hypothesis we can return this median and it is the correct solution to our problem.
The second case is when n is even. Again assuming without loss of generality that $s_s < s_h$ we can conclude as above that the median must be at least as large as s_s. However, in this case we can make the stronger statement that the median is strictly less than s_h. By the definition of the median there are $n/2$ salaries in happytown that are less than s_h and because $s_n < s_h$ there are at least $n/2 + 1$ salaries in sadtown that are less than s_h. Thus there are a minimum of $n+1$ salaries that are less than s_h and s_h is larger than the largest possible median.
Now as above we can consider the subproblem which ignores the $n/2$ salaries less than s_s and the $n/2$ salaries greater then or equal to s_h and the resulting set of salaries will still contain the true median. Furthermore, because we have discarded he same number of salaries below and above the true median the resulting set will have the same median. By the inductive hypothesis we can return this median and it is the correct solution to our problem.

Problem # 4

Problem Description: Return balance scores for a given holiday schedule.
Inputs:

- S = holiday schedule array

Outputs: Hl and Hr. Hl is the number of times a holiday is more popular than a holiday that appears before it and Hr is the number of times a holiday is more popular than a holiday that appears after it.
Assumptions:

- S is a list of n holidays in chronological order. The list contains integers such that S

  $$i$$

  is the number of people that love that holiday.

- No two holidays have the same number of people that love them.

Strategy Overview: I will use a divide and conquer angorithm that repeatedly divides the problem into smaller subproblems.
Algorithm Description:

```
1 Hl = Hr = 0
2
```

```
def sort(lst):
    if (len(lst) == 1):
        return lst

    left = lst[:len(lst)/2]
    right = lst[len(lst)/2:]
    final = merge(sort(left), sort(right))

    return final

def merge(left, right):
    global Hl
    global Hr
    result = []
    while (not (len(left) == 0 and len(right) == 0)):
        if (len(left) == 0):
            result += right
            return result

        if (len(right) == 0):
            result += left
            return result

        if (left[0] <= right[0]):
            Hl += len(right)
            result.append(left[0])
            left = left[1:]

        else:
            Hr += len(left)
            result.append(right[0])
            right = right[1:]

    return result


sort(S)
print "Hl: ", Hl
print "Hr: ", Hr
```

The algorithm is essentially identical to mergesort with an additional constant number of operations to keep track of the scores of Hl and Hr. Therefore the recurrence relation is $T(n) = 2T(n/2) + n$ and the running time is $n \log n$.

Problem # 5

I will divide this algorithm into several parts.

Preprocessing: First sort the collection of lines by increasing slope. As this is done, if two or more lines are encountered that have the same slope, remove all but the one with the highest intercept. This line will at all points dominate the other parallel lines which means they will never appear in the final output.

Divide: The divide portion splits the collection of lines into two subsets based on slope. Thus the left set L contains the half of the lines with the most negative slopes and the right set R contains the half of the lines with the most positive slopes. The lines are repeatedly divided until the base case it reached.
Base Case: The base case is when a collection contains only two lines. To solve the base case we simply calculate the intersection of the two lines. The line with more negative slope is visible to the left of this intersection and the line with the more positive slope is visible to the right. If the collection contains lines ℓ_1, ℓ_2 then the function will return the same collection of lines as well as a collection containing the single point of intersection between the lines.
Merge: The primary challenge in this algorithm is combining the solutions to two subproblems into a solution for a larger problem. Consider the situation at an arbitrary depth of recursion. We have two collections of lines L and R where each set is ordered by increasing slope and the slopes of all lines in L are less than the slopes of all lines in R. Additionally we have a corresponding sets of points P_1 and P_2 where each point is the intersection between two of the lines from the corresponding set of lines. First we note some properties of P_1 and P_2. Due to the ordering of the lines these intersection points are automatically ordered by increasing x coordinate and, given the index of an intersection point, we can find in constant time which two lines it is the intersection of. For example, the first point in P_1 is the intersection between the first two lines in L, the third point is the intersection between the third and fourth lines.

To begin the merging step obtain a pointer to the first point in P_1 and another pointer to the first line in R. While the point dominates the line, increment the pointer to the next line. Eventually one of two things will happen. Either a line will be found that dominates the point, or the list of lines will be exhausted. In the first case we can conclude that the intersection of interest between R and L occurs on line ℓ_1, the first line in L. In the second case we increment the pointer to the next point in P_1. We continue this process and either a point a_i will be found such that there is a line that dominates it, in which case the intersection of interest between L and R occurs on ℓ_i or the list of point is exhausted. In this case the intersection must occur on the last line in L.
We have now identified which line ℓ_i in L is involved in the intersection between L and R. In order to completely determine the point of intersection we need to perform a similar process to discover which line r_j in R is involved in the intersection.
As before, acquire a pointer to the first point in P_2 and a pointer to the first line in L. Repeatedly increment the line pointer until either a line is found which dominates the point or the list of lines has been exhausted at which point increment the point pointer and repeat. As above, either some point b_i will be

found such that there is a line that dominates it, in which case the intersection takes place on line r_{i+1} or all the points will be exhausted which implies that the intersection takes place on the last line in R.
Now, knowing the two lines involved in the intersection we can calculate the point itself. Denote it (x_p, y_p). We need to construct a collection of visible lines and a collection of intersection points to pass up to the next level of recursion. Call the collection of lines F and the collection of points Q. We can begin by adding ℓ_0 to F because we know that the line with the least slope is visible at some point. Then for each (x_i, y_i) in P_1 such that $x_i < x_p$ add (x_i, y_i) to Q and add ℓ_{i+1} to F. Then add (x_p, y_p) to F. Next, for each (x_i, y_i) in P_2 such that $x_i > x_p$ add (x_i, y_i) to Q and add r_2 to Q. Finally add the last line from R to Q.