Question 1

(a) If there are b bees in the swarm at the start of round i, what is the expected number of bees remaining at the start of round i + 1?

Solution:
Given: n bees and n flowers, each bee can choose any flower at random.
Design: For some number p > 0, each bee will attempt to access a flower in each round with probability p, independently of the decisions of the other bees. So, if exactly one bee decides to make the attempt on a particular flower in a given round, it will succeed.

Assume Y as a random variable.
For our case Y = r.v. that is flower with exactly one bee
Goal = find E[Y]
This can be rewritten as a sum of simple r.v's:
Y = X1 + X2 + ... + Xn
where Xi = 1 if flower1 has 1 bee only
else Xi = 0
Expected Values: E[Y]
E[Y] = $\sum_{i=1}^{n} E[X_i] = n * E[X_1]$
= n * (1* P[1 flower with exact 1 bee] + 0 * [else])
= n * (Bees that can sit on flower * P[1 bee on flower 1]*P[rest bees on remaining flowers]
= n * nC1* (1/n) * ((n-1)/n * (n-1)/n *.....*(n-1)/n))
= $bC1 * ((n-1)/n)^{b-1}$ (b total bees)
Remaining bees before round i + 1 =
Total - E[Y]
= b - $b * ((n-1)/n)^{b-1}$

(b) Given: $xj + 1 <= xj^2/n$
Consider, $x2 <= x1^2/n$
$x3 <= x2^2/n$
Substitute value of x2 in terms of x1
$x3 <= (x1^2/n)^2/n$
Generalizing terms $xk + i <= x_i^{2^k}/n^{2^k-1}$
At some point in time at the ith iteration when we use the equation from a, number of bees reduce to half.
$<= (n/2)^{2^k}/n^{2^k-1} <= n/2^{2^k}$
Substituting the above as 1.
we get $n = 2^{2^k}$
$log(n) = 2^k log(2)$
$log(log(n)) = log(2^k) * 1$
Hence Proved,
k = Total rounds = log(log(n))

Question 2

(a) the guest with name-tag i and the guest with name-tag j are wearing matching masks

Solution:
Given: n people and m masks, each person is given a m type mask randomly.

Assume Y as a random variable.
For our case Y = r.v. that is no. of pairs having matching masks
Goal = find E[Y]
This can be rewritten as a sum of simple r.v's:
Y = X1 + X2 + ... + Xn
where Xi = 1 if pair[i,j] has matching masks
else Xi = 0
Expected Values: E[Y]
E[Y] = Probability that i chooses a mask * Probablity that j chooses the same mask * NC2
E[Y] = (1 - Probability that i chooses a mask * Probablity that j chooses some other mask) * NC2
= (1 - (1 * (m-1)/m) * NC2)
= ((1/m) * NC2)
= ((1/m) * (n*(n-1)/2))
= $(n^2 - n)/2 * m$

For what value of n (as a function of m) does this number become 1?
$(n^2 - n)/2 * m = 1$
$= (n^2 - n) = 2 * m$
$= n^2 - n - 2 * m = 0$
On solving this using the quadratic formula we get
n = $(1 + \sqrt{1 + 8m})/2$

(b) Prove that the probability of this having happened (if the template library really has a million designs) is very small
Solution:
Consider the Expected value equation that we got from above:
Substitute m = 10^6 and n = 200
we get E(X) = $(200^2 - 200)/2 * 1000000$
= 0.0199

We can use Markov's inequality eqn to determine how likely is outcome to match the expected value.
Hence,
Pr[Observed value >= t*E[Y]] <= 1/t
= Pr[2 families getting same kits >= t*0.0199 <= 1/t
From this we get t <= 2/0.0199
t <= 100.5
1/t = 0.01
Therefore, The probability of 2 families receiving the same kit from millions of kits is 0.01.

Question 3

(a) Algorithm description
The basic idea is similar to finding the kth smallest element using the pivot method as we did in the class. We would find a random pivot point in A and then create two sub-arrays, one containing elements less than the pivot and the other having more than the pivot. We then calculate the size with the left sub-array and the pivot element. If the size value is present in K (using Binary Search) then we got our kth smallest element for that index. We then remove that element from K and divide K into 2 parts similar to A. We perform recursive calls with left partition A and left partition K. Similarly with the right partitions. And keep on adding the results to our final set.

(b) Consider the following algorithm.

Algorithm 1: Calculate the ki-th smallest number in A for all i = 1, 2, . . . , m

Input: An array A[1..n], containing n elements
An Array K[1..m] containing m elements
Output: Final ResultSet containing kth smallest element for each k in K
`/* Here's my algorithm. */`

```
KSmallestMulti(A[n],K[m]) {
    Create Result R of size m

    if(K is empty) then
        return null
    end

    Pick a random Pivot p from A
    Partition A such that
    A1 <- Elements from A < p
    A2 <- Elements from A > p
    size = A1.size + 1

    if (size in K) then
        remove size from K
        add {size : p} in R
    end

    Partition K such that
    K1 <- Elements from K < size
    K2 <- Elements from K > size

    R.append( KSmallestMulti(A1, K1) )
    Subtract size from each element in K2
    R.append( KSmallestMulti(A2, K2) )    // Add size to index while appending

    return R
}
```

(c) Correctness:
Consider the example given in question: A = {1, 5, 9, 3, 7, 12, 15, 8, 21}, K = { 2, 5, 7 }
Suppose we take the random pivot as 8.
A1 = 1,5,3,7 and A2 = 9,12,15,21
size = 4 (A1.size) + 1 = 5
We have 5 in our K hence we add 5: 8 in our result set (indicating 5th smallest element is 8)
Now recursively call the same method for 1,5,3,7 with K = { 2 } and 9,12,15,21 with K = { 2 (7 minus 5) } We would get the 2nd smallest element as 3 and the 7th smallest element (2nd smallest in the right subarray) as 12.

(d) Running time analysis.
T(n) = n + 2T(n/2,m/2)
T(n) = n + n + 4T(n/4,m/4)
T(n) = n + n + + n + 8T(n/8,m/8)
Similarly
T(n) = k*n + $2^k T(n/2^k, m/2^k)$
$m/2^k = 1$
k = logm
Hence on substituting,
T(n) = O(n*log(m))

Question 4

(a) Algorithm description
The basic idea is similar to finding the kth smallest element using the pivot method as we did in the class. We would find a random pivot point in A and then create two sub-arrays, one containing elements less than the pivot and the other having more than the pivot. We then calculate the size with the left sub-array and the pivot element. If the size value is present in K then we got our kth smallest element for that index. We then remove that element from K and divide K into 2 parts similar to A. We perform recursive calls with left partition A and left partition K. Similarly with the right partitions. And keep on adding the results to our final set.

(b) Consider the following algorithm.

Algorithm 2: find the largest element x in A such that the sum of the elements of A less than x is at most M

Input: An array A[1..n], containing n elements
M = target sum
Output: Largest element with leftmost elements having at most sum M

```
/* Here's my algorithm.  */
```

```
kthSmallestSum(A, M)
{
    Pick a random Pivot P
    Partition A such that
    A1 : set of elements of A < P
    A2 : set of elments of A > P

    sum = sum of Elements in A1

    if (sum == M) then
        return p
    else if (sum > M)
        return kthSmallestSum(A1, M)
    else
        if (sum + P) > M    // Adding pivot exceeds M
            return P
        else
            return kthSmallestSum(A2, M-sum-p)
        end
    end

}
```

(c) Correctness:
Consider the example given in question: A = {4, 8, 5, 2, 3, 6, 1}, M = 10
Suppose we take the random pivot P as 6.
A1 = 4,5,2,3,1 and A2 = 8
sum of A1 = 15 15 ¿ 10
we repeat same for A1
Assume random pivot 3
A1 = 2,1 and A2 = 4,5
Sum of A1 = 3
3 ¡ 10 Go to A2 and M = 10 - 3 - 3 = 4
Consider pivot 5 for 4,5
A1 = 4 and A2 = empty set
A1 sum = 4
Since sum = M, hence return pivot = 5
Answer = 5

(d) Running time analysis.
Time complexity here would be similar to the kth smallest element done in class where we solved the recurrence relation $T(n) = T(Max\{|A1|, |A2|\}) + n$ and got O(n).