Solution - 3

1.

Under the hash function h(x) = x mod 10, all the keys collide.

If I randomly choose my hash function from the universal hash function family we discussed in class I expect 115 and 2545 to collide with probability 1/10. I randomly chose: a = 4253, and b = 580.

What really happened was (4253*115 + 580) mod 10007 mod 10 = 9 and (4253*2545 + 580) mod 10007 mod 10 = 8. There was a 1 in 10 chance they collided, and for this choice of a and b they did not collide.

Using this same hash function on the other keys:

(4253*995 + 580) mod 10007 mod 10 = 1

(4253*5025 + 580) mod 10007 mod 10 = 0

I had no collision.

Expected number of collisions <= 1000/2000 = ½, since there are already a 1000 keys present in the hash table.

2.

Using corollary 11.11 and 11.12 from the text book

a. P [using more 8n space]
$= P\left[E\left[\sum_{i=0}^{m-1}(n_j)^2 > 8n\right]\right] \leq \frac{2n}{8n} = \frac{1}{4}$
So the probability is 1-1/4 = ¾

b. P [using more kn space]
$= P\left[E\left[\sum_{i=0}^{m-1}(n_j)^2 > kn\right]\right] \leq \frac{2n}{kn} = \frac{2}{k}$
So the probability is 1-2/k = (k-2)/k

3. When we store n keys in a hash table of size m=n 2 using a hash function h
randomly chosen from a universal class of hash function, the probability is less than 1/2 that there is any collision. Therefore, probability that there are no collisions is larger than 1/2. So,
p(n) > 1/2 + (1/2) 2 + (1/2) 3 + ... + (1/2)n > 99.9999%
=> 1 - (1/2)n > 99.9999%
=> n > 19
=> n = 20
So, we try 20 times.

4.

No; For example, x1 = 1, x2 = 6, m = 5, Pr[h(x1) = h(x2)] = 1

5. Good News ! Since we covered hashing in lecture 3, we now onwards have the incredibly powerful hash-based structures (dictionary, hashset, hashtable etc) available to us to use as subroutines.

Two pieces go together if they weigh w, so w1 +w2 = w means the pieces go together. Let us use "weights" to represent the array containing all these pieces.
We will parse through the array once and put each of the items into a hash-structure. We will also calculate the frequency of the pieces if we see similar pieces. This parses "weights" once and so in O(n).

Now we will go through the keys in the HashMap and for each key w1 find the “opposite” corresponding piece (w - w1) (using get/search O(1)) and delete that element (O(1)). W1 and w2 are now two valid pieces that can be put back together. In a worst-case scenario, where each piece was unique we will iterate over n/2 “w1” pieces, and find and delete (O(1)) their corresponding n/2 pieces “w2”. This makes the algorithm O(n)

6. At the core of this question is the sorting algorithm “counting sort” which you can learn more about here: https://en.wikipedia.org/wiki/Counting_sort

Assumptions: The input consists of an array of recruit *objects* that needs to be sorted according to their rank, which we can find from a recruit object’s rank attribute, the highest rank “*k*” and “n” the size of the input array. For example, if “x” is a recruit object, “x.rank” would be the rank of “x”.

```
SORT_BY_RANK (array recruits, integer k, integer n)
    Initialize array output of size n            // the range of output is 1 to n inclusive.
    Initialize array counter of size k           // the range of counter is 1 to k inclusive.
    for i = 1 to k                               // initialize the counter array to 0
        counter[i] = 0

    for i = 1 to n                               // count the number recruits in per rank
        current_recruit = recruits[i]
        counter[current_recruit.rank] += 1

    cumulative_sum = 0
    for i = 1 to k:                              // find the cumulative sum of counter
        cumulative_sum += counter[i]
        counter[i] = cumulative_sum

    for i = 1 to n
        current_recruit = recruits[i]
        index = counter[current_recruit.rank]  // find the index of current_recruit from counter
        counter[current_recruit.rank] -= 1    // decrement counter[rank] so that the next time we
                                              //encounter the same rank, it gets index - 1
        output[index] = current_recruit          // set the output value

    return output
```

Time complexity analysis: There are 4 loops. 2 of them loop for k times and the other 2 loop for n times. All other code takes constant amount of time. Thus, the time complexity is O(2n+2k) = O(n+k)

7.

```
class recruit():
    def __init__(self, name, colony):
        self.name = name
        self. colony = colony


def sort_colonies(input,c,n):
    result=[]
    for i = 1 to n:
        if(result[input[i].colony] != NULL):
            result[input[i].colony].count+= 1;
        else:
            result[input[i].colony].id = input[i].colony;
            result[input[i].colony].count= 1;
    Mergesort(result);
    return result;
```

Building the result array costs O(n) and mergesort costs O(clogc), thus the total cost is O(n+clogc)

8.

a. (5 points) Probability that a person on guest list was put in prison is 0. Becase the hash function always generate the sample hash value for the same input. Therefore, if a person is on the guest list, the value on his position on the hash table must be 1.

b. (7 points) In order for a person who was not on the guest list to get into the party, its hash value must collide with someone who was invited. Therefore, this question is asking us to compute the probabililty of collision.

Let m be the size of the hash table. Let n be the number of guests invited to the party.

For a person who is not on the guest list to not be allowed in they must not hash to the same location as any of the n guests who are on the guest list: $(1-1/m)^n$

For them to be allowed in, this is $1-(1-1/m)^n$

c. (8 points) Given that the table size remains the same. In this case, for every guest i, we will use two hash functions.

$v1 = h1(i)$

$v2 = h2(i)$

Then we will set both T[v1] and T[v2] to 1. When a new guest shows up, we will accept him/her only if both T[v1] and T[v2] is 1.

For a person not on the guest list to have T[v1] = 0, the chance they do this is $(1-1/m)^{2n}$, since now 2n items have been hashed into the table with the simple uniform hashing assumption.

So for a person not on the guest list to have T[v1] = 1, the chance is $1 - (1-1/m)^{2n}$.

The same is true for v2.

For both T[v1] and T[v2] to both be 1 it must be:

$(1- (1-1/m)^{2n})^2$

FYI, this technique is called bloom filter. You can learn more about it here: https://en.wikipedia.org/wiki/Bloom_filter