1. $h(115) = 5, h(2545) = 5, h(995) = 5, h(505) = 5, there\ are\ three\ collisons$

   $$h_R(x) = \big((2x + 1) mod 10007\big) mod 10$$

   The probability that the keys 115 and 2545 collide is not greater than 1/10.

   $$h_R(115) = 1, h_R(2545) = 1, h_R(995) = 1, h_R(505) = 1, three\ collisons$$

   The expected number of collisons is not greater than 2000/1000=2.

2. $\Pr[\sum_{i=0}^{m-1}(n_i)^2 \geq 8n] \leq \frac{E[\sum_{i=0}^{m-1}(n_i)^2]}{8n} < \frac{2n}{8n} = \frac{1}{4}$

   $$\Pr\left[\sum_{i=0}^{m-1}(n_i)^2 \geq kn\right] \leq \frac{E[\sum_{i=0}^{m-1}(n_i)^2]}{kn} < \frac{2n}{kn} = \frac{2}{k}$$

3. because the probability of no collisions is greater than $\frac{1}{2}$, so when I attempt once, the failure rate is less than 1/2, when I attempt twice, the failure rate is less than 1/4, so n is the times that I attempt.

   $\left(\frac{1}{2}\right)^n < \frac{1}{1000000}$ n≥20.

4. It is not universal.

   Assuming that k=ax mod m, l=ay mod m, x and y are two different key.

   If k=l, then a(x-y) mod m =0, because $a \in [1, m-1]$, x-y must be times of m.

   In that way, if x=5, y=10, m=10, no matter what a is, x and y will always collide.

5. I define a hash table S in this algorithm, and the key of S is the weight of pieces and the value of S is the index of pieces with the weight. The same weight hashes to the same slot. If there are two pieces with the same weighs, use chaining to solve collisions.

```
def    function(weights)
Hash_table S
for i=1 to weights.length
      if( search(S, tw-weights[i]))      // if there exists a piece weighing w-weights[i]
            j=search(S, w-weights[i])      //return the list of S[w-weights[i]]'s head pointer
            delete(S,j)                       //delete pointer j from the list
            print    j and i    belong together
      else
            Insert( S,i)
```

The loop goes n times, and in average case Search, Delete and Insert's running time are O(1), so the algorithm's average running time is O(n).

6. In this algorithm, I will use two level hash table whose running time is O(1) in worst case. I hash the recruits with the same rank into the same slot, then hash them in the second level, finally traverse the hash table from key =1 to key =k.

```
def function(recruits)
    Two level hash table S
    for i=1 to n
          Insert( S, recruits[i])
   for j=1 to k
          Search(S, j)
         Print all recruits who are ranked j.
```

Because the two level hash table has the running time of O(1) in worst case, so in the first for loop, it will cost O(n), in the second for loop, it will cost O(k), so the algorithm's running time is O(n+k) in worst case.

7.In this algorithm, I will use two level hash table S to solve this problem. In the first level of hash table, two recruits from the same colony hash to the same slot, and then use another hash function to hash those recruits from the same colony. After the first hashing, I can know the number of every colony's recruits. Finaly I use a MAX heap to sort those colonies.

```
def function(recruits)
Two level hash table S
for i=1 to n
     Insert( S, recruits[i])
A=Getsize(S.firstlevel) //A is an array which stores every colony's recruiter number.
BUILD_MAX_HEAP(A)
for j=1 to c
     m=MAX_EXTRACT_HEAP(A)
     print colony m provides the jth most recruiters.
```

Two level hash table has O(1) in worst case. So the running time of hashing is O(n) in worst case. BUILD_MAX_HEAP's running time is O(n).And MAX EXTRACTHEAP will cost O(logc) every time, there are c colonies, so the heap sort's running time is O(clogc). Therefore, the algorithm's running time is O(n+clogc).

8. Assuming the table size of T is m.

The probability that a person on the guest list was put into prison is $\frac{1}{m}$.

The probability that a person who wasn't on the guest list was allowed into the party is $\frac{1}{m}$.

Yes, I can use two level hash table with two different hash function. After the first hash function, I can know how many items hash to the same slot in the first level, then the second level table size is the square of the number in first level. In this way, we could reduce the number of collison effectively. And the space is also $\Theta(n)$, so we do not need to enlarge the table size to implement this algorithm.