**ANALYSIS FOR TRAFFIC FINES**

1. Find out the drivers who are booked for legal action:

[Hash tables](http://en.wikipedia.org/wiki/Hash_table) are **O(1)** average and [amortized](http://en.wikipedia.org/wiki/Amortized_analysis) case complexity, however it suffers from **O(n)** worst case time complexity. Hash tables suffer from worst time complexity **O(n)** due to two reasons:

1.     If too many elements were hashed into the same key: looking inside this key may take time **O(n).**

2.     Once a hash table has passed its [load balance](http://en.wikipedia.org/wiki/Load_balancing_%28computing%29) - it has to rehash [create a new bigger table, and re-insert each element to the table].

However average O(1) and amortized case because:

1.     It is very rare that many items will be hashed to the same key [if we chose a good hash function and we don't have too big load balance.

2.     The rehash operation, which is O(n), can at most happen after n/2 operations, which are all assumes O(1) : Thus, when we sum the average time per operation, we get: **(n \* O(1)  +  O(n)) / n)  =  O(1)**

Direct hashing is the worst way to perform hashing in terms of space used. Also, there is a notion of collision. (When more than one data lands on a same index.) Chaining is the most preferred way to resolve the collision where we would attach a data structure such as a linked list or a tree to that with major hashing algorithms such as MD5, SHA.

We would rather say that **O(1)** with higher probability. Chaining can make it **O(n) or O(log N)** depending on the datastructure used

2. Find out the policemen who are eligible for bonus:

We have the fine amount collected by the policemen represented as binary search tree.  The time complexity of the binary search algorithm belongs to the **O(log n)** class.  The interpretation is that the asymptotic growth of the time the function takes to execute given an input set of size n will not exceed **log n.** When n grows very large, the log n function will out-grow the time it takes to execute the function. The size of the "input set", n, is just the length of the list.

We are just eliminating half of the elements to be searched for until we find the element we need. Say initially we have N number of elements and then what we do is ⌊N/2⌋ as a first attempt. Where N is sum of lower bound and upper bound. The first-time value of N would be equal to (L + H), where L is the first index (0) and H is the last index of the list we are searching for. If we are lucky, the element we try to find will be in the middle [e.g. we are searching for 24 in the list {6, 18, 24, 36, 48} then we calculate ⌊(0+4)/2⌋ = 2 where 0 is lower bound (L - index of the first element of the array) and 4 is the higher bound (H - index of the last element of the array). In the above case L = 0 and H = 4. Now 2 is the index of the element 24 that we are searching found.

If the case was a different array{6, 18, 20,24, 36, } but we were still searching for 24 then we would not be lucky, and we would be doing first N/2 (which is ⌊(0+4)/2⌋ = 2 and then realize element 24 at the index 2 is not the number we are looking for. Now we know that we don’t have to look for at least half of the array in our next attempt to search iterative manner. Our effort of searching is halved. So basically, we do not search half the list of elements that we searched previously, every time we try to find the element that we were not able to find in Our previous attempt.

for(i=0;i<n;n=n/2)

{

i++;

}

1. Suppose at i=k the loop terminates. i.e. the loop executes k times.

2. at each iteration n is divided by half.

2.a n=n/2                   .... at i=1

2.b n=(n/2)/2=n/(2^2)

2.c n=((n/2)/2)/2=n/(2^3)....... at i=3

2.d n=(((n/2)/2)/2)/2=n/(2^4)

So, at i=k , n=1 which is obtain by dividing n  2^k times

n=2^k

1=n/2^k

k=log(N)  //base 2