REPORT:

- The following assumptions have been made in the following answers as per the Assignment sheet:
    1. Separate chaining is the collision management method.
    2. When a collision is identified the elements in the ‘bucket’ will be linked in the order they arrived, that is the first item in the linked list is the first item recorded and the nth item to arrive is at depth n in the list.
    3. The variable **size** refers to the number of possible ‘buckets’ in the hash table.
    4. The variable **n** refers to the total number of elements to be hashed.
    5. Operations such as assignment and comparison of integers and pointers are constant time.
    6. For 1 - 3 the type of hash function is unknown and so the complexity of producing the hash will not be taken into consideration and will as such be treated as constant.

1. Assume all elements are hashed to the same bucket:
    - i. Insert time for this type of hash will grow larger as more items are inserted into the table as a longer and longer linked list will need to be traversed to find an element’s final destination as described by the assumptions above. For this reason the insertion of the nth item of the hash table will require O(n) complexity to be achieved, considering the necessary assignments in this process as constant operations.
    - ii. Search time likewise will be just as complex, knowing that the elements are insert.ed in order of arrival, to find the nth item inserted will require traversing a linked list of effective length n and so will require O(n) to be achieved, any item before the nth will be associated with an effective list length of some fractional multiple of n and so can have its search also described as O(n).

2. Assume elements spread evenly throughout buckets, that is each bucket contains an equal portion of the elements.
    - i. This set up will only have a proportional increase in efficiency as the number of elements per bucket will be decreased however the overall complexity model will remain O(n) for an insert operation. We know that each bucket contains an equal number of elements and that in a table of size buckets and n items this means there will be n/size elements in every bucket. Due to this insertion of the nth item into a bucket will take O(n/size) complexity, treating size as a constant this is O(n) however to retain generality we will state the complexity as O(n/size).
    - ii. As insertion occurs in order of arrival to find an element, perform a search operation, will take the same time as the insertion and so in this case will be O(n/size) retaining generality.

3. Assume no different inputs share a hash, or in essence for a set of unique elements there will be no collisions and so only one element per bucket, the below analysis will refer to the situation in which all elements are unique for simplicity.
    - i. Knowing in this situation there are no collisions insert becomes a very simple operation. Every time an element is hashed it is assigned a bucket, no lists need to be traversed as such the insertion of an item is as simple as pointer assignment inside an array, as stated in the opening assumptions pointer assignments are taken as constant and so the complexity of this operation is O(1).
    - ii. In order to perform search an element need only be hashed and an array element called, this is a constant operation and so search has complexity O(1).

4. The function bad_hash in module hash.c is an extremely ineffective hash as it uses a constant value and only one piece of information to create a hash for an element, this reduces the amount of 'randomness' in the hash. Although a is random it is necessary that it is static for consistency and so creates some dangerous properties for this hash function the first of which relates to the numeric properties of a itself. In a situation where rand() produces an a that just so happens to be an integer multiple of size the uniqueness of the key[0] values passed to bad_hash become irrelevant as the expression a*key[0] becomes an integer multiple of size as well and will always become 0 in mod size. This means that every element of the hash table will hash to 0 creating a situation as described in 1. This kind of situation could also occur if every element in the input began with the same letter making the expression a*key[0] the same for all elements and therefore producing the same hash every time.

5. Assuming sufficient randomness in the selection of r and a large enough size it is reasonable to assume size independent variables make up the r factors, with repetition given the maximum possible length of the input is larger than size. If we assume in the average case the input doesn't exceed size in length then it is reasonable to assume each variable used in the hashing process is unique and so lends sufficient randomness to the eventual hashes. Given this it should be expected that an even spread of elements are hashed across the table given n elements where n > size. In this event we would expect a situation similar to that of question 2 and so search and insert processes should be in the order of complexity O(n/size).

6. Explanation and analysis of collide_dumb.
   i. My algorithm for the collide_dumb function uses a greedy input approach, iterating through all one character strings in the ASCII decimal range 33 - 126 inclusive (this excludes all formatting characters and the extended series of ASCII to avoid rendering issues in the output). Each of these candidate strings are hashed with universal_hash and the output hash examined, if it is 0 this string is recorded if not the next one is examined until n strings are found that hash to 0. In the event not enough one character strings hash to 0, one of the recorded one character strings is copied and has characters appended to it to find 2 character strings that hash to 0 and if still not enough strings found this is repeated ad infinitum until n strings are recorded (important to note string length will not exceed MAXSTRLEN).
   ii. In order to find 2 strings that hash to 0 with this collide_dumb algorithm lets assume both of these solution strings are one character and so only one loop is required and no string copies are made, this simplifies the complexity consideration somewhat. This means that the loop will only run until the last of the 2 is found, giving the second solution's ASCII decimal designation the name s2 we then have that the loop will run O(s2 - 33) times, remembering all character who's ASCII designation is less than 33 have been discarded. Therefore the complexity to find 2 solutions is O(s2). The value of s2 will most likely be greater than 2 in most cases and so this algorithm is extremely inefficient.

3. Explanation and analysis of collide_clever.
   i. The algorithm I designed for collide_clever works very similarly to that used in collide_dumb however instead of simply searching for a needle in a haystack it uses modular arithmetic to create known solutions and record them. Like collide_dumb in the event that not enough single character solutions can be constructed it has a multi-character contingency which appends characters using the same analysis used to create the single character strings. The mathematical principle behind my algorithm is quite simple, knowing that the random factors used in universal_hash are publicly available and understanding the method which this function uses to create hashes it is possible to design candidate solutions as integers and then cast them as unsigned char unsigned to create solution strings. The objective of this collision function is to create hash to 0 solutions, for universal_hash this means that every character in the solution string, when multiplied by it's corresponding random factor must be 0 in mod size so

that the summation of these will also be 0 in mod size and satisfy universal_hash (recall (a+b)mod size = (a mod size + b mod size)mod size)). This premise implies that the product of r_i and s_i (where r_i and s_i are the ith random factor and corresponding character in a string) must be an integer multiple of size, as only integer multiplies of size will be 0 mod size, mathematically that is $r_i * s_i$ = m*size, where m is an integer. Knowing that r_i is an integer already and that the integers are closed under multiplication this simplifies to s_i = k size, k is an integer. With this knowledge my function iterates through all integer multiples of size that are less than ASCII decimal 127 and only records those larger than ASCII decimal 32, these are then recorded as solutions and hashed just as a precaution to check accuracy. These ASCII boundary conditions mean that only solutions in the inclusive range ASCII decimal 33 -126 will be used in order to avoid rendering issues as discussed in the above explanation of collide_dumb.

ii. There are two possible situations for the complexity of this algorithm and they depend on how many one character solutions are available compared to n. As the algorithm relies on integer multiples of size in the ASCII range 33-126, the number of characters that satisfy this will be 126/size - 33/size, lets call this figure m. If m > n (the number of solutions wanted) then it holds that all solutions will be one character and so the complexity will be O(n) as the iteration cycle will only need to run n times. However if m < n this means string copies will have to be made for the multi-character contingency which will inevitably increase complexity. To copy a string is O(len) where len is the length of the given string, in terms of the variables used in the function Num_randoms is the length of a given string so if m < n the complexity of the algorithm becomes O(n) + O(Num_randoms - 1) as the string being copied is one character less than the current. This evaluates to O(n) + O(Num_randoms).