DESIGN OF ALGORITHMS

ASSIGNMENT 2

Shreyash Patodia | 767336

University of Melbourne | SEMESTER 1, 2016

Shreyash Patodia | 767336

Report: Assignment 2 –What makes a good or bad hash function

In this report I answer the questions posed to us as we progressed through Assignment 2 for Design of Algorithms.
The report refers to list and bucket interchangeably

Question 1:

If the hash function hashes everything to the same bucket then the complexity of insertion and search will be independent of the size of the hash table, *size* and will only depend on the number of inputs, *n*.
Insertion: Assuming we add elements to the list such that every new item added to the list will be added at it's end, we would potentially need O(n) time for each insertion. If we just added items to the front of the list we would just need O(1) time for each insertion.
Thus, for the first type of insertion (to the end of the list) we have $O(n^2)$ total time for insertion and for insertion to the start of the list we have O(n) time corresponding to the n items that are to be inserted.
Search: If we want to search for an item in the list it would be O(n) irrespective of the method of insertion. We would thus need $O(n^2)$ in order to look for n items in our list.
(We can have O(1) search for an item but that is the best case and might not be a good indicator for the running time of the algorithm)

Question 2:

If the hash function spreads the input perfectly evenly over all the buckets then we may have to think about the number of inputs, *n* relative to the size of the hash table, *size*.
Each bucket would have n/size number of elements.
Insertion: Strictly speaking if we insert new items to the end of the list and we have a function that spreads the inputs evenly then if the size and n are of the same order (or n < size) then we would need O(1) insertion for each item and a total O(n) time for n items. If the n gets large compared to size (For example, $n \sim size^2$) then we would stop having constant insertion times and would have insertion depending on O(n/size) for each item and for n items it would simply translate to $O(n^2/size)$. We stop assuming constant behavior since we cannot assume that the length of each list is trivial as n gets larger as compared to the size of the hash table.
Search: Search in the lists would also depend heavily upon the ration of n/size if the n is of the same order as the size then the search for an item would be constant time O(1) (thus, O(n) for n items) due to the low population of the lists but as n gets larger and larger, the search time for an item would be better described by O(n/size) and thus, a search of n items would potentially take $O(n^2/size)$.

Question 3:

If the hash function never hashes two things to the same position, then it would mean that number of inputs must be less than the size of the hash table otherwise we would have to

hash two things to the same value or increase the size of the hash table. So, we will only talk about the hash function assuming n < = size.
Insertion: Insertion of an item in the list would trivially be O(1) since we are guaranteed to have an empty list when we hash something to it. Thus, an insertion of n items would take O(n) time.
Search: Due to the fact that the list at each hash value will trivially contain at most one value we will have O(1) search times and thus, in order to be able to search for n items we would need O(n) time in total.

Question 4:

The hash function so described would be a bad hash function because we would be determining the hash for a given string using only the first letter of the string, implying that if the input contains strings that all start with the same letter then our hash function would cause our hash table to decompose into a fancy linked list. For example if our input was all the cities in the world with names beginning with M the search would be O(n) and insertion of an item to the end of the list would potentially be O(n) if we insert at the tail of the list or O(1) if we insert to the start.
Example:

- Melbourne
- Madrid
- Mumbai

Question 5:

We know that if we implement a Universal Hashing function the probability of a collision is 1/size where size is the size of the hash table. This is similar to the hash function in Question 2 that spreads the input evenly. As long as the number of inputs is comparable to the size of the hash table our insertion and search operations for an item would be expected to be O(1) and thus O(n) for n items. But as, n gets larger we would need need to mind the ratio of n/size and insertion and search for an item would become O(n/size). Thus, leading to $O(n^2/size)$ insertion and search for n items. (This is the same as question 2 where we had the bucket size = n/size).
The universal hash function itself runs in constant time which depends on the length of key that needs to be hashed by it, in any case since the length of the key cannot be more than MAXSTRLEN we can safely assume that the universal hash function runs in constant time.

Question 6:

In this question n is the number of collisions to be generated
I will attempt to step through my collide_dumb() function that I implemented for Assignment-2:

- Declaration and initialization of all the variables in the function takes constant time and we can classify it as O(1). The printing of all the random variables can also be regarded as O(1).
- We then go on to try to generate collisions using the characters in the domain string (contains all the alpha-numeric characters). We do this by iterating through a loop as long as either on of two things happen: we generate the required number of strings or we are not able to generate any more strings

- Now we generate each string, a string can have a maximum length equal to MAXSTRLEN – 2 (we try to mind 2 spaces for the null byte and the new line char), so the generation of a string is cheap enough to be regarded as O(1) time.
- After the generation of a string we check if it hashes to 0, we do this by calling the universal hash function which can also run in time proportional to the length of the string to be compared and thus we can regard it as O(1) due to existence of MAXSTRLEN.
- Now we spend some extra time in order to check if a string which we have found to be hashing to zero is a string that was previously generated or not. To check this we need to do comparisons with an array of strings which may potentially contain O(n) strings, thus if we assume strcmp() to be O(1), we can say that O(n) time is spent on checking whether the current string that hashes to zero was previously generated or not
- We repeat the above process MAX_TRY_PERLEN = 1000000 times for each length of the string, this number may seem too large and wasteful but because the number of combinations for a string of length n increases exponentially, the definition of MAX_TRY_PERLEN to a very large value seems justified in order to cover a large enough number of strings to cover all possible values.
- Now we know that we basically spend O(n) time (from string comparisons) on each string and the probability of a collision is 1/size, where *size* is the size of the hash table. Thus we will need to generate (on average) size number of strings in order to generate a collision (from the expectation of a geometric variable where the success would be defined as a collision and failure is lack of collision).
- Thus, we expect to have time proportional to O(n*size) in order to find one collision. in order to find n collisions, we would thus potentially need $O(n^2 size)$ time.

Thus, we would need to have O(4*size) comparisons which is just O(size) number of comparisons in order to generate 2 collisions to 0. However, if n is larger then it cannot be ignored and needs to be taken into account.

Question 7:

In this question n is the number of collisions to be generated

In order to show what motivated my method for collide_clever(), we will keep in mind the simple math trick that:

LowestCommonMultiple(a, b) = (a* b)/GreatestCommonDivisor(a*b)

Now with this result at hand we want to generate strings such that they hash to zero when multiplied to our random numbers. Let us start off by generating a single character that hashes to zero when multiplied by the first random number, now we know that the LCM (Lowest Common Multiple) of size and the random number will tell us this value but we also know that the GCD of any random number and the size is 1, so size*random number is the LCM of the random number and the size. Thus, we can just multiply the random number by the size to get something that hashes to zero. We can extend this idea to strings and multiply all the random numbers by the size of the hash table in order to make it hash to

zero (Note we find a multiple of the table size greater than 32 if our hash table size is less than 32).
Since all we are doing is multiplying our random number by our size we take O(1) operations to find one collision. If the size of the hash table is less than 32 we will take at most 6 operations in order to find a multiple greater than 32 so it can also be regarded as an O(1) operation. Since, we need to generate n strings we would need to do the same operations described above n time, giving us a overall complexity of O(n).
0 cannot be the size of the hash table