Hash analysis

For this analysis, I used 100 fake data to analyse. The key of the fake data is random adjective words generated by google. So the key is not totally random because they are under the same part-of-speech. This data is found in the hash_analysis.py file and ALL time complexity mentioned is referring to the time complexity of __linear_probe method. So this analysis is to see how good hash and bad hash affects the time complexity of a hash table to search, insert or delete an item.

<u>Good hash function:</u> (my group used universal hashing)

The variable a and b work in a way so that the (hash base/coefficient) for every position of the key is different and so called 'random'. Although it is not truly random but this is good enough to generate sparse data and we cannot use a truly random number generator as this will hash the keys that are the same to different positions. I predict that the good hash function will have a relatively low chance for conflict to happen because the chance of collision is very low, 1/tablesize. The total probe length will also be much lower compared to bad hash because this function should avoid cluster formation and the max probe length will be low. If max probe length is low, the time complexity of inserting an item is more likely to be O(1). All the above is true when table size is at least double the number of items in the hash table.

<u>For this section, table size is (number of items)*2, in this case 200. When it is prime, tablesize is 199.</u>

When a and b is a prime (a=31397, b=27179) and tablesize is prime, the statistics() returns (29,49,8)

When a and b is a prime (a=31397, b=27179) and table size is not a prime, the statistics() returns (30,61,9)

Provided that table size is not equal to (a or b). When either (a and b) or table size is a prime, they will be coprime which shares no common factors, that's why there is a small difference in the results. However, when both (a and b) and table size are prime, the results are slightly better because the key will be hashed to more 'random' positions. This is because when a and b are not prime, they will share common factors, and the values will not be very spread out.

When a and b is not a prime (a=31415, b=27185) and tablesize is prime, the statistice() returns (32,76,10)

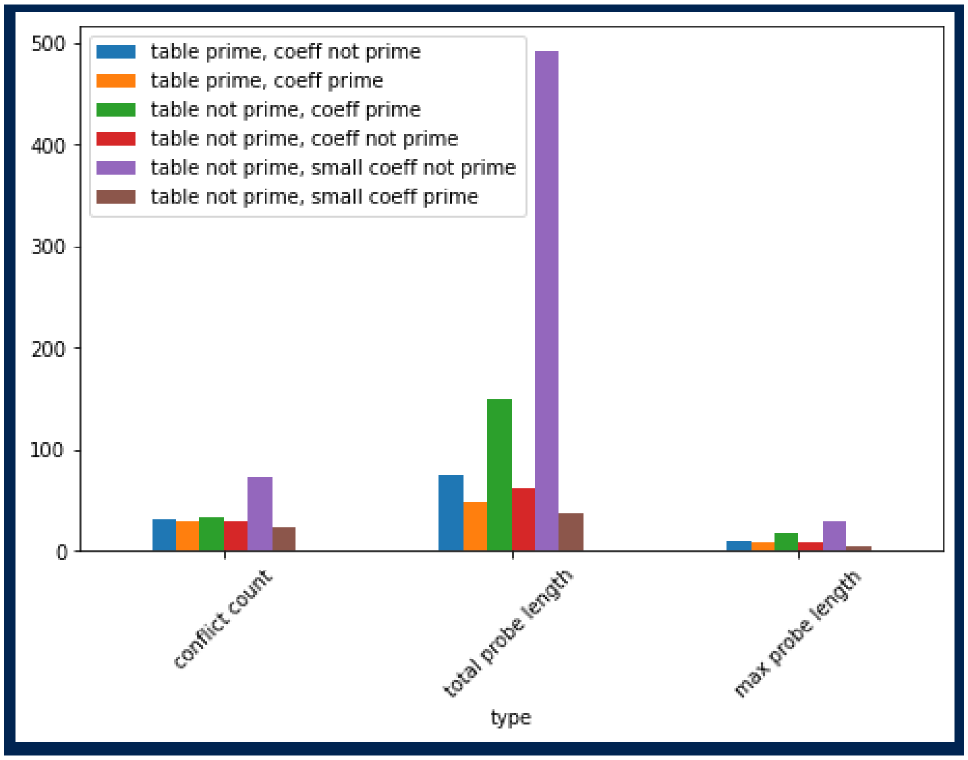
When a and b is not a prime (a=31415, b=27185) and tablesize is not a prime, the statistics() returns (33,149,18)

For the first result, although coefficients and tablesize are coprime but when coefficients are not prime, they will have common factors and the value will not be very spread out. For the second result, we can clearly see that the total probe length is higher than others. This is because there will be common factors and more keys will be hashed to the same position and cause clustering. Clustering tends to grow larger once its formed.

However, based on the above observations, (except the last case), the maximum probe length is still relatively small and probably can still be considered as O(1) for its time complexity. That is because I used universal hashing that will generate 'random' coefficients for every positions and the value of a and b is relatively big compared to tablesize.

When only 1 small non-prime coefficient used(a=20), and tablesize is not prime. The statistics() returns (73, 493, 30)

When only 1 small prime coefficient used (a=19), and tablesize is not prime. The statistics() returns (23, 37, 4)



Based on the graph plot, the best case is the orange bar where tablesize and (a and b) are both prime. The worst case is the purple bar where tablesize is not a prime and coefficient is small and not a prime. Although the max probe length of the worst case is not close to O(n) where n is table size but the use of correct values will greatly improve the efficiency of the runtime. This is because the chances of collision happening will be reduced. The chances of conflict happening is around 30% when the right values are used.

In conclusion, the conditions that avoid clustering and generate spread out values are:

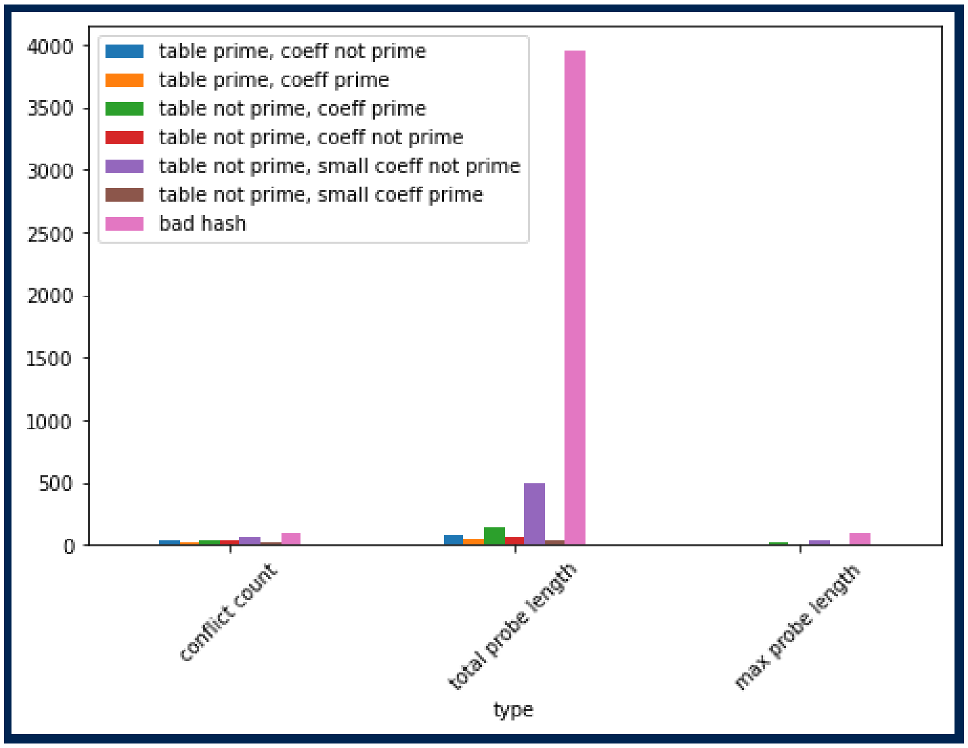
1. Prime table size
2. Coefficients are prime and relatively large compared to tablesize
3. Table size and coefficient is coprime
4. Use 'random' coefficients for every position of key

Bad hash function

For our bad hash function, we only consider the ord value of the last character in the key and we make sure that table size is not prime by using largest_prime(tablesize) + 1 as final tablesize. My prediction is that the conflict count and max probe length count will be very close to the total number of items in the hash table. This is because every key that has the same last letter will be hashed to the same position so the chances of collision and conflict happening is very high. The max probe length will also be very close because once a cluster forms, it tends to grow and the chances of collision is high.

The statistics() will always return (91,3963,98). The time complexity is O(n) where n is the number of items in the hash table. It is never O(n) when n is tablesize, this is because this bad hash function will form a cluster that is almost as long as the total number of items so linear probe will always serach through the whole cluster instead of the whole table. O(n) where n is table size, will probably occur when there is rehashing and resizing of the hash table but I did not analyse on this part for this assignment.

The graph when bad hash is included.



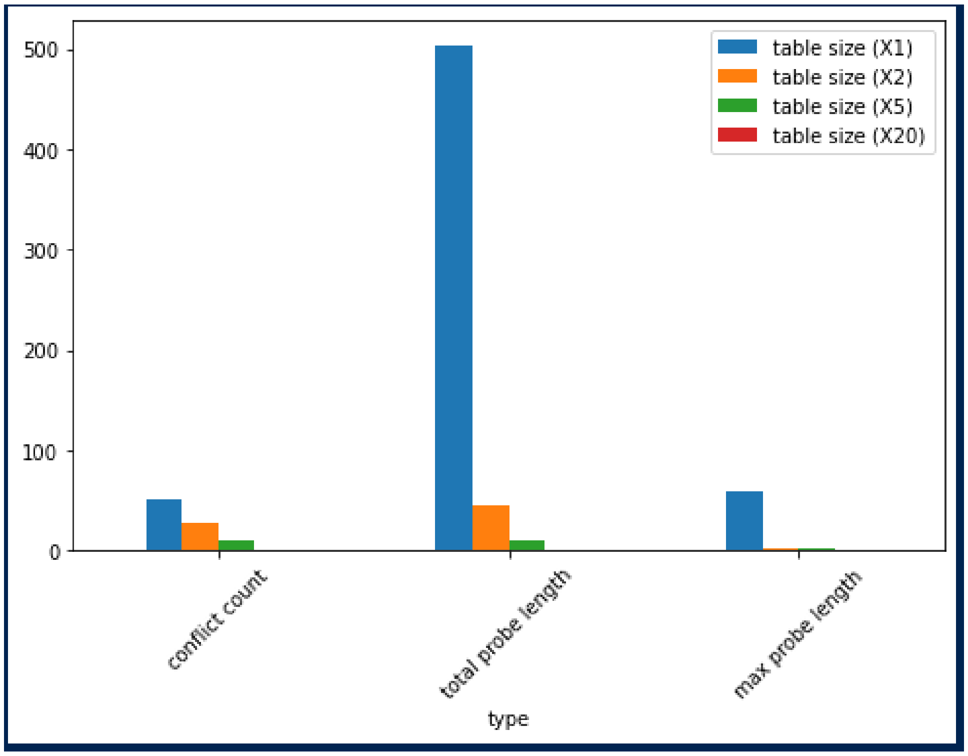
Load Factor

When table size is equal to the number of items in hash table, the statistics() returns (51,504,60)

When table size is double to the number of items in hash table, the statistics() returns (28,45,3)

When table size is 5 times larger compared to the number of items in hash table, the statistics() returns (10,11,2)

When table size is 20 times larger compared to the number of items in hash table, the statistics() returns (0,0,0)



The larger the table size, the lower the load factor. But using a table size that is 20 times the number of items is too large and uses extra space. So, base on the graph, we can see that the main difference is when load factor is 1 and load factor is 0.5. The max probe chain reduced from 60 to 3. So the tendency of clustering occurs when the load is more than half of the tablesize. In conclusion, I decided to use a table size that is 2 times the number of items in this assignment as it avoids clustering and a bigger size does not improve the time complexity even more so its not necessary to use extra space.