# Hash Tables
## Software Engineering for Scientists

https://github.com/swe4s/lectures/tree/master/src/hash_tables

In many problems, we want to store and retrieve data by a key-value pair. Arrays are useful data structures for storing values because you can directly access the data, assuming you know the index of the key. Suppose we have the array `county_population` that holds pairs with county name and population

```
county_population = [('Boulder',294567), ('Denver',600158), ('Summit',27994)]
```

and we want to know the population for Boulder County. If we know the index of `Boulder` in `county_population`, then we can simply reference that position. If we don't know the index of `Boulder`, then we can either search `county_population` to retrieve the index, which can be slow, or we can use a **hash function** ( `h(key)=index` ) that maps each county name (the key) to an index in `county_population`. Together, a hash function and an array make a **hash table**. Suppose we have a hash function `h()` where

```
h('Boulder') = 0      h('Denver') = 1       h('Summit') = 2
```

Then we can easily and quickly get the population for `'Boulder'`

```
county_population[h('Boulder')][1]
```

which seems nice, but has a major drawback.

To avoid any confusion, `h()` must be a **perfect hash**, which means that no two keys can ever hash to the same index ( `h(a) == h(b)` if and only if `a == b` ). The issue is that the space required to store values in a perfect hash is proportional to the number of keys and not the number of values. For example, if we want to store the counties for just two samples `'Boulder'` and `'Grand'` where `h('Boulder')=0` and `h('Grand')=999` then the `county_population` array must have at least 1000 elements.

A more practical approach is to specify the size of the value array and use the modulo function (`%` in python) to cast the output of the hash function (which can be nearly unbounded) to a valid index of the value array. Suppose the `county_population` array has 5 elements and we want to store the `('Boulder',294567)` and `('Grand',14843)`. By taking `h(key)%5`, we can fix the unbounded size issue.

| key | value | h(key) | h(key) % 5 |
|---|---|---|---|
| 'Boulder' | 294567 | 0 | 0 |
| 'Grand' | 14843 | 999 | 4 |

```
county_population = [('Boulder',294567), , , , ('Grand',14843)]
```

The issue with this strategy is that collisions can occur. For example, inserting `('Eagle',52197)` where `h('Eagle')%5 = 4` will collide with `'Grand'`. Assuming we are using a **uniform hash function** where the likelihood that two keys hash to the same index is $\frac{1}{N}$ where $N$ is the size of the value array (e.g., $N = 5$ for `county_population`), then the likelihood of a collision is proportional to the **load factor** $\frac{M}{N}$ where $M$ the number of keys inserted into the array (e.g., $M = 3$ for the previous example).

While this ratio shows that the likelihood of a collision can be reduced by increasing the size of the array, it is intractable to prevent collisions. Given the uniform hash, the likelihood that any two keys hash to the same index is $\frac{1}{N}$. After inserting $M$ keys, the expected number of collisions is $\binom{m}{2}\frac{1}{N} = \frac{M(M-1)}{2N}$. This is a depressing result. As the number of keys in the array grows linearly the number of collisions grows quadratically, while the number of collisions decreases linearly as the size of the table grows linearly. To keep the number of collisions low, $N$must be much larger than $M$.

In practice, hash table implementations will limit the number of collisions by doubling their size ($N = 2N$) when the load factor passes some predetermined threshold (e.g., 0.7). This process is called **rehashing** since all of the keys that were in the array must be rehashed with the new array size.

## Linear Probing

Since increasing the size of the array only limits the collisions, various strategies have been proposed to deal with collisions.

A simple strategy is to start at the hashed index, then scan the array until the correct index is found. For insertion, the correct index is the first open slot. For search, it is the index with a matching key. With linear probing, inserting `('Boulder',294567)`, `('Grand',14843)`, then `('Eagle',52197)` produces:

| key | value | h(key)%5 | county_population |
|---|---|---|---|
| 'Boulder' | 294567 | 0 | [('Boulder',294567),,,,] |
| 'Grand' | 14843 | 4 | [('Boulder',294567),,,,('Grand',14843)] |
| 'Eagle' | 52197 | 4 | [('Boulder',294567,('Eagle',52197),,,('Grand',14843)] |

The first collision occurs when attempting to insert `'Eagle'` at index 4, which is occupied by `'Grand'`. Since 4 is at the end of the array, the next index to be considered is 0, which is also not empty. Finally, index 1 is empty and `('Eagle',52197)` is stored there. As the load factor increase, the time required to insert increases.

Searching for `'Boulder'` and `'Grand'` end at the hash position 0 and 4 because the keys stored in those slots match. Searching for `'Eagle'` begins at index 4, then moves on to 0 and 1 because the keys don't match. At index 1 the keys match and the search completes. Similar to insertion, as the load factor increases search time increases.

## Chained hash

Another strategy to deal with collisions is by appending the key-value pair to an array at the hashed position. With chained hashing, inserting `('Boulder',294567)`, `('Grand',14843)`, then `('Eagle',52197)` produces:

| key | value | h(key)%5 | county_population |
|---|---|---|---|
| 'Boulder' | 294567 | 0 | [[('Boulder',294567)],,,,] |
| 'Grand' | 14843 | 4 | [[('Boulder',294567)],,,,[('Grand',14843)] ] |
| 'Eagle' | 52197 | 4 | [[('Boulder',294567)],,,, [('Grand',14843),('Eagle',52197)] ] |

Since every insertion occurs at the hashed position and always involves appending to the end of a list, the time required is independent of the load factor. Searching for a key involves scanning the list at the hashed index for a matching key. The average length of each list is equal to the load factor ($\frac{M}{N}$), so as the load factor increases search times also increase.

## Hash functions

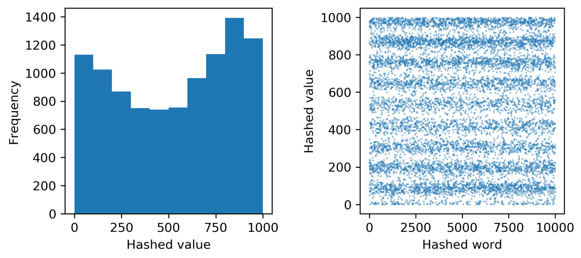
To get the index that corresponds to a key, we need a hash function. While the only requirement for correctness is that the hash is deterministic. (`h(a) == h(a)` for every `a`), an efficient hash function will produce values that are as close to uniformly distributed as possible. The more uniform the hash function is the close the actual probably of collisions will be to $\frac{1}{N}$, which is the best-case scenario.

In principle, hash functions can operate on any type of data, from strings and integers to lists. Here we will only consider strings. The object of these functions is to take a string and a table size as input and return an integer that is less than the table size. For each of these functions, we analyze the uniformity of the output consider both random strings (`rand_words.txt`) and non-random strings (`non_rand_words.txt`).

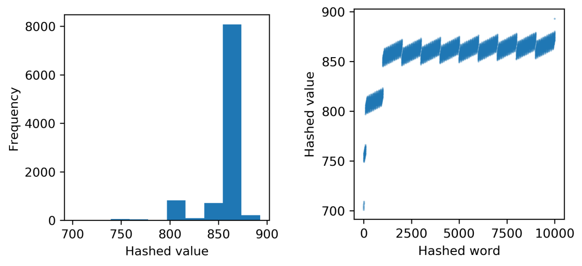
ASCII hash: This is a very basic hash function that uses the `ord()` function covert characters to their integer representation.

```python
def h_ascii_sum(key, N):
    s = 0
    for i in range(len(key)):
        s += ord(key[i])
    return s % N
```

Random



Non-random

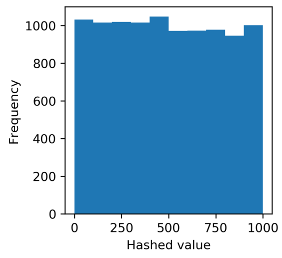


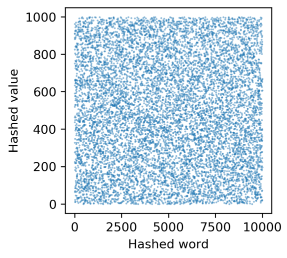
Rolling hash: A slightly more complex approac where

- p is a prime number roughly equal to the number of characters in the input alphabet
- m should be large, since the probability of two random strings colliding is about 1/m. $2^{64}$ is a good choice.

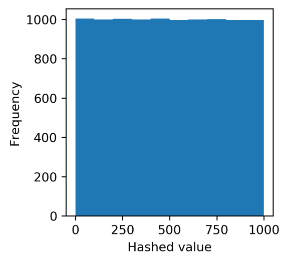
```
def h_polynomial_rolling(key, N, p=53, m=2**64):
    s = 0
    for i in range(len(key)):
        s += ord(key[i]) * p**i
    s = s % m
    return s % N
```

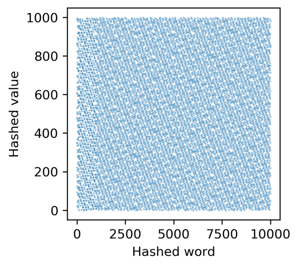
Random





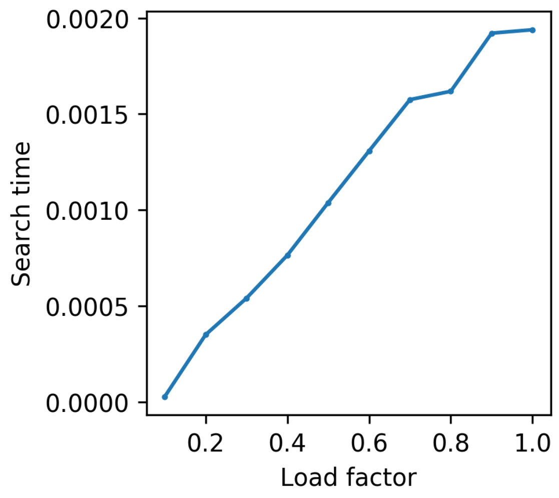
Non-random



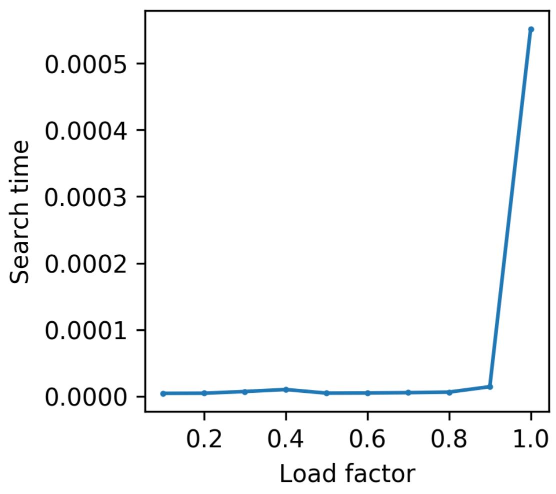


These differences in the uniformity of these two hash functions directly affect the runtime efficiency of lookups in the hash tables

ASCI hash



Rolling hash


Search time
Load factor