Hash Functions and Chaos

Ambareesh Shyam Sundar

4 December 2023

What is a hash function?

What is a hash function?

- Converts input of arbitrary length into fixed-length output

What is a hash function?

- Converts input of arbitrary length into fixed-length output
- Frequently used in hash tables

Hash function: `hash(key) = key % 17`

key	hash(key)	value
5	5	"hello"
8	8	"goodbye"
36	2	"compression"
22	5	"collision"

Desirable properties of hash functions

Desirable properties of hash functions

- Deterministic (`a == b` and `hash(a) == X` imply `hash(b) == X`)

Desirable properties of hash functions

- Deterministic (`a == b` and `hash(a) == X` imply `hash(b) == X`)
- Non-invertible (given `X == hash(a)`, it is difficult to determine the value of `a`)

Desirable properties of hash functions

- Deterministic (`a == b` and `hash(a) == X` imply `hash(b) == X`)
- Non-invertible (given `X == hash(a)`, it is difficult to determine the value of `a`)
- Resistant to collisions (given `hash(a) == X`, it is difficult to find `b != a` such that `hash(b) == X`)

Desirable properties of hash functions

- Deterministic (`a == b` and `hash(a) == X` imply `hash(b) == X`)
- Non-invertible (given `X == hash(a)`, it is difficult to determine the value of `a`)
- Resistant to collisions (given `hash(a) == X`, it is difficult to find `b != a` such that `hash(b) == X`)
- Avalanche effect (given `a` and `a'` are slightly different, even if `hash(a) == X` is known, it is hard to predict `hash(a')`)

Desirable properties of hash functions

Avalanche effect

Desirable properties of hash functions

Avalanche effect

$\Updownarrow$

Sensitive dependence on initial conditions

Desirable properties of hash functions

Avalanche effect

$\Updownarrow$

Sensitive dependence on initial conditions

$\Updownarrow$

Chaos?

Desirable properties of hash functions

Avalanche effect

$\Updownarrow$

Sensitive dependence on initial conditions

$\Updownarrow$

Chaos?

(transitivity helps)

The big question:

Can chaotic discrete maps be used as effective hash functions?

For this experiment:

For this experiment:

- Hash strings (`char[]`) to 32-bit unsigned integers (`unsigned int`)

For this experiment:

- Hash strings (`char[]`) to 32-bit unsigned integers (`unsigned int`)
- 4 different hashing algorithms
  - Rolling hash
  - Tent map
  - Dyadic map
  - “Multi-state” hash

For this experiment:

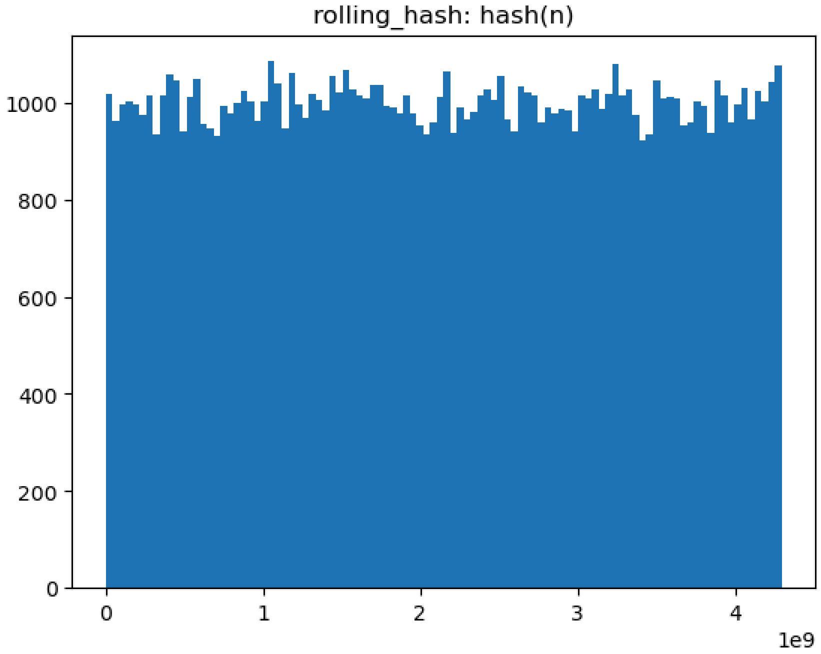
- Hash strings (`char[]`) to 32-bit unsigned integers (`unsigned int`)
- 4 different hashing algorithms
    - Rolling hash
    - Tent map
    - Dyadic map
    - “Multi-state” hash
- 3 different metrics
    - Distribution of hashed values
    - Number of collisions
    - Avalanche effect

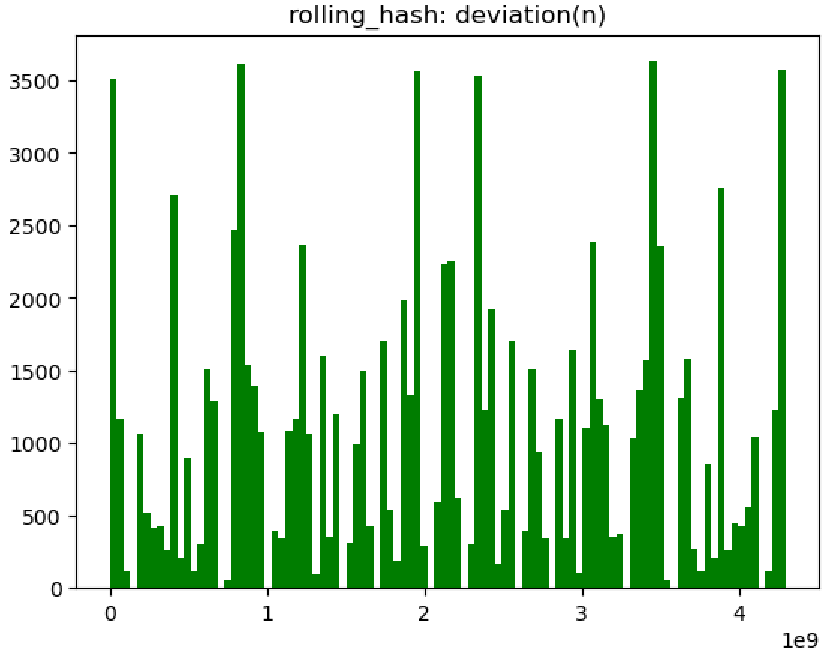
Rolling hash

```c
unsigned int rolling_hash(char *n) {
    unsigned int hash = 0;
    unsigned int curr_pow = 1;
    for (unsigned long i = 0; i < strlen(n); i++) {
        hash += n[i] * curr_pow;
        curr_pow *= 7919;
    }
    return hash;
}
```

(Interpret the string as an integer in base 7919.)

Rolling hash





Collision percentage: 0.003%

Tent map hash

$$T(x) = \begin{cases} 2x & \text{if } x < \frac{2^{32}-1}{2} \\ 2 \cdot ((2^{32} - 1) - x) & \text{otherwise} \end{cases}$$

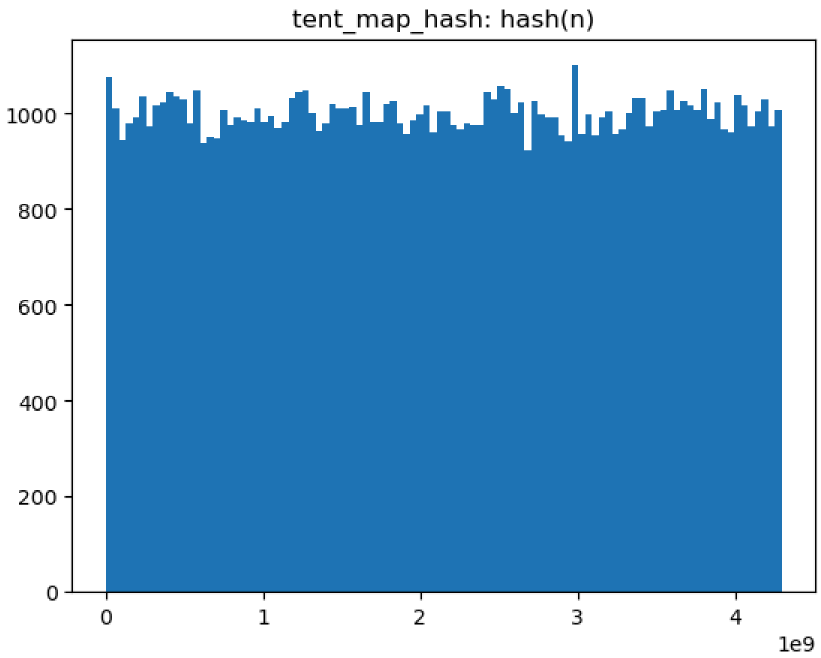
```
unsigned int tent_map_hash(char *n) {
    unsigned int hash = rolling_hash(n);
    int iterations = 53;
```

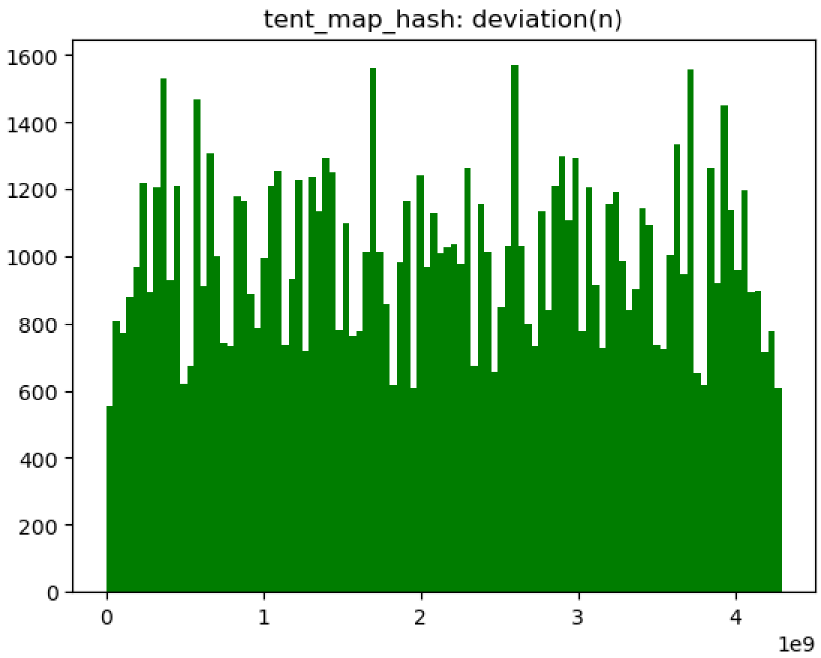
(Find the rolling hash value of the string.)

```
    for (int i = iterations; i > 0; i--) {
        if (hash < UINT_MAX / 2) {
            hash = 2 * hash;
        } else {
            hash = 2 * (UINT_MAX - hash);
        }
    }
    return hash;
}
```

(Perform 53 iterations of the tent map on the hash value.)

Tent map hash





Collision percentage: 0.001%

Dyadic map hash

$$D(x) = \begin{cases} 2x & \text{if } x < \frac{2^{32}-1}{2} \\ 2 \cdot \left(x - \frac{2^{32}-1}{2}\right) & \text{otherwise} \end{cases}$$

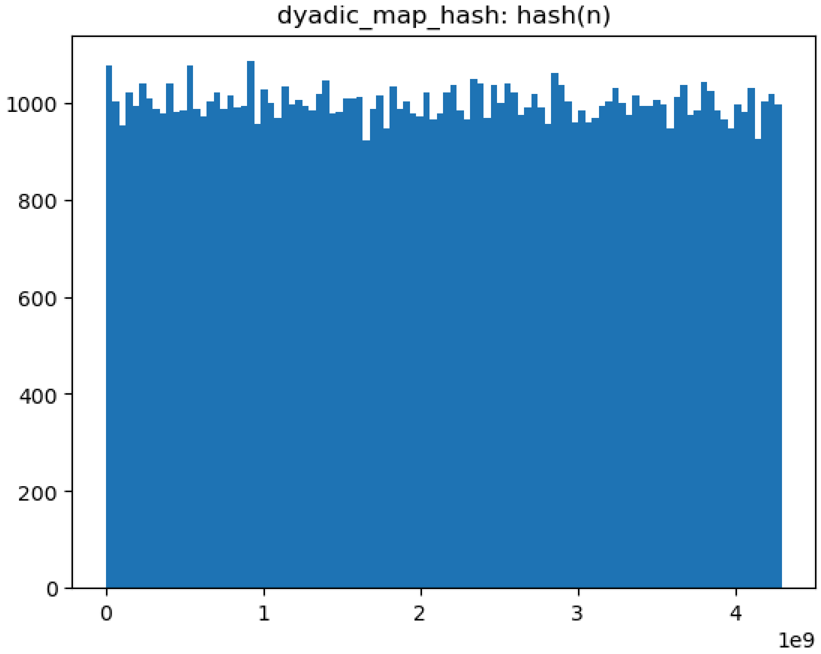
```
unsigned int dyadic_map_hash(char *n) {
    unsigned int hash = rolling_hash(n);
    int iterations = 53;
```

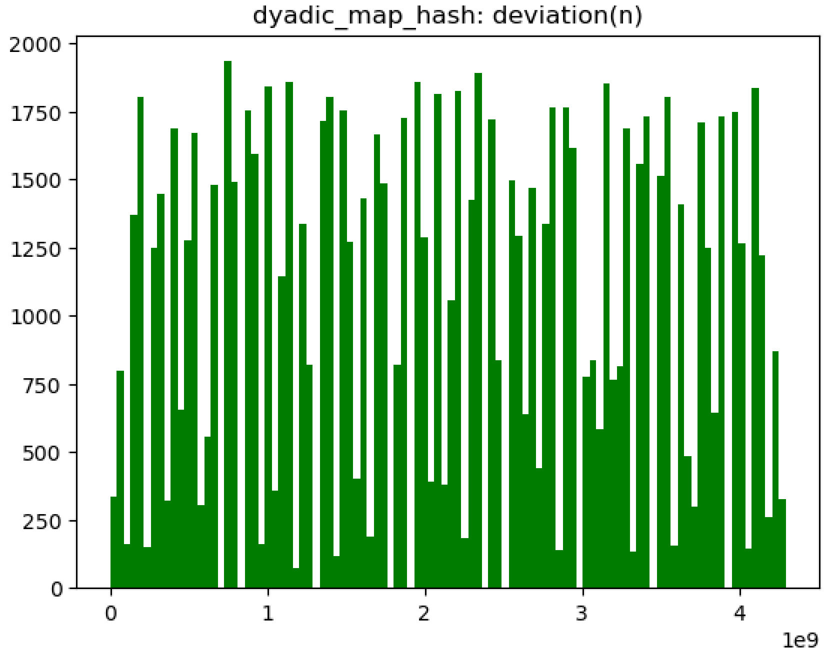
(Find the rolling hash value of the string.)

```
    for (int i = iterations; i > 0; i--) {
        if (hash < UINT_MAX / 2) {
            hash = 2 * hash;
        } else {
            hash = 2 * (hash - UINT_MAX / 2);
        }
    }
    return hash;
}
```

(Perform 53 iterations of the dyadic map on the hash value.)

Dyadic map hash





Collision percentage: 0.002%

The dyadic map and tent map

- Suspiciously similar hashes between tent map and dyadic map

The dyadic map and tent map

- Suspiciously similar hashes between tent map and dyadic map
- Only for certain strings

The dyadic map and tent map

- Suspiciously similar hashes between tent map and dyadic map
- Only for certain strings

```
tent_map_hash with input "asdfqwer": 914563734
tent_map_hash with input "asdfqwgr": 1406886406
tent_map_hash with input "asdfqwer.": 4153872816

dyadic_map_hash with input "asdfqwer": 914564396
dyadic_map_hash with input "asdfqwgr": 1406886924
dyadic_map_hash with input "asdfqwer.": 141098142
```

The dyadic map and tent map

- Topological conjugacy:

  $$T(x) = C^{-1}(D(C(x)))$$

  where

  $$C(x) = \sin(\pi x)$$

The dyadic map and tent map

- Topological conjugacy:

  $$T(x) = C^{-1}(D(C(x)))$$

  where

  $$C(x) = \sin(\pi x)$$

- If $C(x) = \sin(\pi x) \approx x$, then $T(x) \approx D(x)$

What can we improve?

What can we improve?

- Rolling hash is weak

What can we improve?

- Rolling hash is weak
- Easy to find collisions
  - Degrades performance due to lots of accidental collisions

What can we improve?

- Rolling hash is weak
- Easy to find collisions
  - Degrades performance due to lots of accidental collisions
- Not chaotic, so small perturbations $\longrightarrow$ small changes in hash
  - Transfers over to tent map and dyadic map hashes

What can we improve?

- Rolling hash is weak
- Easy to find collisions
  - Degrades performance due to lots of accidental collisions
- Not chaotic, so small perturbations $\longrightarrow$ small changes in hash
  - Transfers over to tent map and dyadic map hashes
- **Solution:** remove dependence on rolling hash

MD5 algorithm

MD5 algorithm

- Maintain four state variables: A, B, C, D (each 1/4 of the output)

MD5 algorithm

- Maintain four state variables: A, B, C, D (each 1/4 of the output)
- Pad input in order to break it into chunks

MD5 algorithm

- Maintain four state variables: A, B, C, D (each 1/4 of the output)
- Pad input in order to break it into chunks
- For each chunk:
  - Apply a (chaotic) map on (B, C, D) and store it in A
  - Rotate all values ($A \longrightarrow B$, $B \longrightarrow C$, …)
  - After the chunk is processed, add the values of A, B, C, D to the global state

MD5 algorithm

- Maintain four state variables: A, B, C, D (each 1/4 of the output)
- Pad input in order to break it into chunks
- For each chunk:
  - Apply a (chaotic) map on (B, C, D) and store it in A
  - Rotate all values (A $\longrightarrow$ B, B $\longrightarrow$ C, …)
  - After the chunk is processed, add the values of A, B, C, D to the global state
- Return A concat B concat C concat D

Multi-state hash

Multi-state hash

- Maintain four state variables: A, B, C, D (each an 8-bit integer)

Multi-state hash

- Maintain four state variables: A, B, C, D (each an 8-bit integer)
- Pad input to be a multiple of `32 * 4 = 128` bits

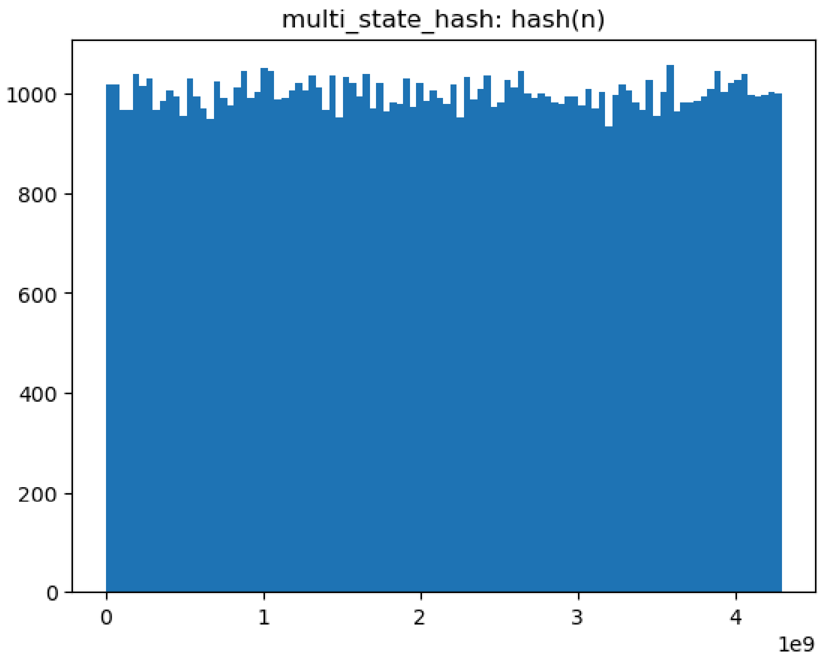
Multi-state hash

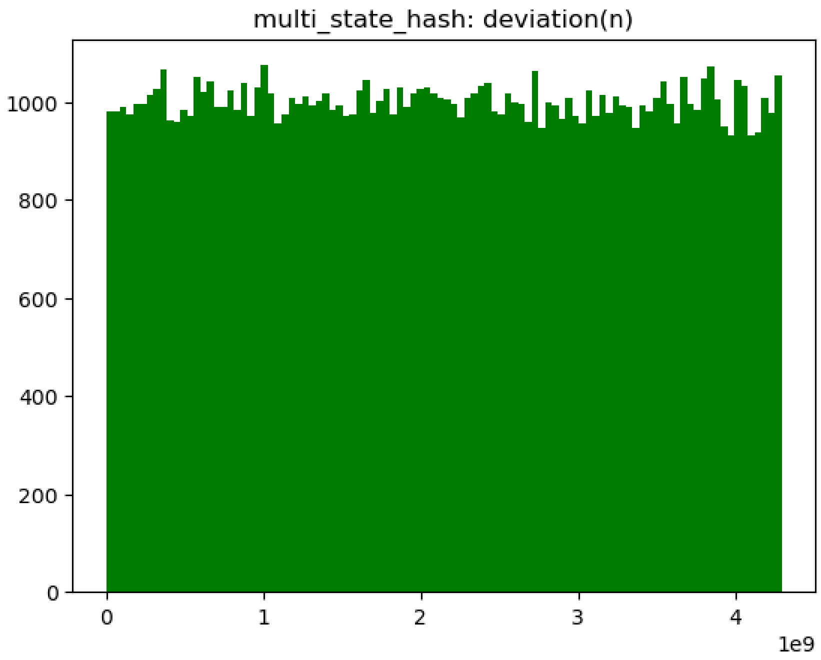
- Maintain four state variables: A, B, C, D (each an 8-bit integer)
- Pad input to be a multiple of `32 * 4 = 128` bits
- Add the next 8-bit chunk of input to A, apply the tent map to A, then rotate state values
  - Repeat 32 times per 8-bit chunk

Multi-state hash

- Maintain four state variables: A, B, C, D (each an 8-bit integer)
- Pad input to be a multiple of `32 * 4 = 128` bits
- Add the next 8-bit chunk of input to A, apply the tent map to A, then rotate state values
  - Repeat 32 times per 8-bit chunk
- Return A concat B concat C concat D

Multi-state hash





Collision percentage: 1.162%

Conclusions

Conclusions

Can a chaotic map be used to construct a simple and effective hash function?

Conclusions

Can a chaotic map be used to construct a simple and effective hash function?

Yes!

Conclusions

Can a chaotic map be used to construct a simple and effective hash function?

Yes! (mostly)

Conclusions

Can a chaotic map be used to construct a simple and effective hash function?

Yes! (mostly)

Things that worked:

- Determinism
- Hash distribution with multi-state hash
- Avalanche effect

Conclusions

Can a chaotic map be used to construct a simple and effective hash function?

Yes! (mostly)

Things that worked:

- Determinism
- Hash distribution with multi-state hash
- Avalanche effect

Things that didn't really work:

- Collision resistance

Potential future research:

- How can collision resistance for the multi-state hash be improved?
- How much padding is really necessary for the multi-state hash to be effective?

Thanks for listening!

Source code for hashes + ipynb of analysis available at

github.com/ambareesh1510/chaotic-hash-functions