Lecture 10.
Hashing.

Algorithms and Data Structures
Ivan Solomatin
MIPT 2020

Outline

- Hash basics
  - Hash definition
  - Universal hash
  - Universal families
- Rabin-Karp algorithm
  - Idea
  - Rolling hash
  - Implementation
- Hash table
  - Idea
  - Separate chaining collision handling (Linked list)
  - Open addressing collision handling

Hash basics

- **Hash definition**
- Universal hash
- Universal families

Hash definition

Definition

Hash is just a function which maps set of keys to a (usually) smaller set of values.

$$h: K \to V$$

$$|V| < |K| \text{ (usually)}$$

Usually:

$$V = \{0, 1, \dots, M - 1\}$$

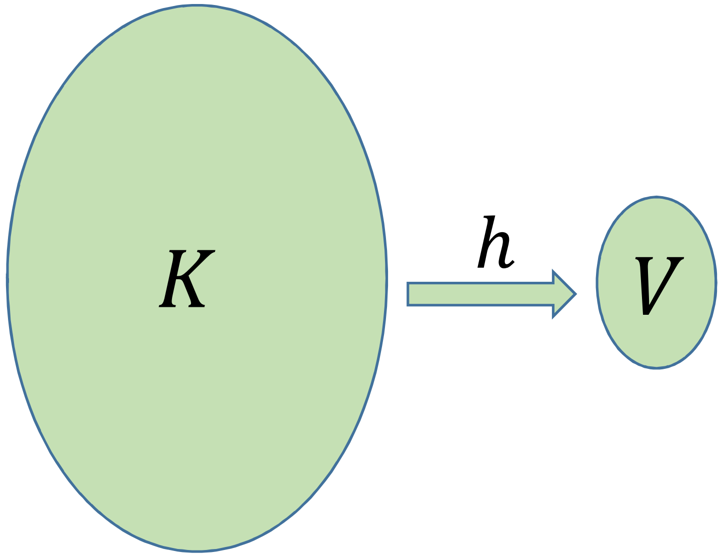
E.g.:

K – set of strings

$$h('aba') = 10$$

$$h('daba') = 7$$

…



Hash definition

Definition

As soon as $|V| < |K|$, h is a surjective function.

$$k_1 = k_2 \Rightarrow h(k_1) = h(k_2)$$

But,

$$\exists k_1, k_2 \in K\colon \big(k_1 \neq k_2 \cap h(k_1) = h(k_2)\big)$$

It is called **hash collision**.

Indicator value for collision between two keys:

$$C_h(k_1, k_2) = \begin{cases} 0, & if\ k_1 = k_2 \cup h(k_1) \neq h(k_2) \\ 1, & if\ k_1 \neq k_2 \cap h(k_1) = h(k_2) \end{cases}$$

Hash definition

Usage

How can we use hashes? We can use them to speed up comparison of complex objects (e.g. strings).

Let’s suppose that we have list of strings $S = ['a', 'b', 'abacaba', \dots]$.

And also we have list H with their hash values: $H[i] = h(S[i])$

Let’s suppose that we need to check if given string x is in S: $x \overset{?}{\in} S$.

Let’s denote: $|S| = N, |x| = l, V = \{0, \dots, M-1\}$

Naïve solution will do that (in worst case) in $O(Nl) -$ we need to explicitly compare each string from S with x.

But we have information about hashes, which gives us additional opportunity to make fast pre-check for negative cases:

$$H[i] \neq h(x) \;\; \Rightarrow \;\; S[i] \neq x$$

But if $H[i] = h(x)$, we cannot guarantee that $S[i] = x$ and we need to compare them explicitly.

Number of operations will be $O\big(N + l(\sum_i C_h(S[i], x) + 1)\big)$.

Hash definition

Usage

Number of operations will be $O\left(N + l(\sum_i C_h(S[i], x) + 1)\right)$.

To minimize complexity we need to minimize $\sum_i C_h(S_i, x)$. To achieve best worst-case complexity we need to minimize maximum possible number of collisions:

$$\max_{\substack{x,\, S: \\ |x|=l, |S|=N}} \sum_{s \in S} C_h(s, x) \to \min$$

But as soon as we fixed h, we can choose S to be a subset of set of any N preimages of $h(x)$ not equal to x, in other words, set of keys which collide with x:

$$S_x = \{s \in K:\ h(s) = h(x)\} \setminus \{x\}$$

$$\sum_{s \in S \subseteq S_x} C_h(s, x) = N$$

Each s collide with x

That drops our complexity to initial one:

$$O(Nl)$$

Let's try to fix that.

Hash basics

- Hash definition
- **Universal hash**
- Universal families

Universal hash
Idea

The problem we experienced in the previous part is quite similar to one we had in QSort: when we fix a procedure in our algorithm (in QSort we fixed pivot choice), in set of all possible inputs, we can find an input which drops complexity of the algorithm.

In QSort we used random pivot choice to handle that.

Let’s try to apply randomness to our hashes.

Universal hash

Idea

Let’s suppose that our function h is not a fixed one, but a random function from set of all possible functions (sampled uniformly):

$$F = \{f : K \to V\}; \qquad h \sim U(F)$$

In this case, $\sum_{s \in S} C_h(s, x)$ is a random value. Let’s analyze mathematical expectation of complexity of our method:

$$\mathrm{E}(T) = O\left(\mathrm{E}\left(N + l\left(\sum_{s \in S} C_h(s, x) + 1\right)\right)\right)$$

Universal hash

Idea

$$\mathrm{E}(T) = O\left(\mathrm{E}\left(N + l\left(\textstyle\sum_{s \in S} C_h(s,x) + 1\right)\right)\right) =$$
$$O\left(N + l + l\,\mathrm{E}\left(\textstyle\sum_{s \in S} C_h(s,x)\right)\right)$$

$$\mathrm{E}\left(\sum_{s \in S} C_h(s,x)\right) = \sum_{s \in S} \mathrm{E}\left(C_h(s,x)\right) = \sum_{s \in S} P[C_h(s,x) = 1] =$$
$$\sum_{s \in S} P(s \neq x \cap h(s) = h(x)) = \sum_{s \in S} \frac{1}{M} = \frac{N}{M}$$

So, if we choose $M \geq N$:

$$E(T) = O\left(N + l + \frac{Nl}{M}\right) \leq O(N + 2l) = O(N + l)$$

Universal hash

Idea

The problem of this approach is that random function is very hard to represent in computer memory.

Amount of possible functions:

$$|F| = |\{f: K \to V\}| = |V|^{|K|} = M^{|K|} \text{ – enourmosly big}$$

So, we can’t store random function which is uniformly sampled from F.

But we can try to take smaller set of functions $H \subset F$ and parametrize it. That will allow to store parameters instead of values for each key. That leads us to idea of **universal families.**

Hash basics

- Hash definition
- Universal hash
- **Universal families**

Universal families

Definition

A family of functions $H \subseteq \{f : K \to V\}$ is called **universal** if:

$$\forall x, y \in K: P_{h \in H}[C_h(x, y) = 1] \leq \frac{1}{|V|} = \frac{1}{M}$$

$$\forall x, y \in K:\ x \neq y: P_{h \in H}\big(h(x) = h(y)\big) \leq \frac{1}{|V|} = \frac{1}{M}$$

Example: for integer numbers: $K = \{0, 1, \dots, |K| - 1\}$.

One should choose a prime number first: $p \geq |K|$.

$$h_{a,b}(x) = \big((ax + b)\ mod\ p\big)\ mod\ M$$
$$H = \{h_{a,b}: a \neq 0 \cap a, b \in [0; p)\}$$

H is a universal family.

Universal families

Integer numbers

Let’s **prove** this family to be universal.

$$h_{a,b}(x) = \big((ax + b)\ mod\ p\big)\ mod\ M$$
$$H = \{h_{a,b}: a \neq 0 \cap a, b \in [0; p)\}$$

Actually, each function consists of two transformations:

$$\begin{tikzcd} K \arrow[r, "(ax+b)\ mod\ p"] & K' \arrow[r, "mod\ M"] & V \end{tikzcd}$$

$$|K| \leq |K'| \gg |V|$$

Let’s analyze whether collision is possible in first transformation:

$$(ax + b)\ mod\ p \equiv (ay + b)\ mod\ p$$
$$a(x - y) \equiv 0\ \ (mod\ p)$$
$$x - y \equiv 0\ \ (mod\ p)$$
$$x = y$$

p – prime ⇒
no zero divisors

$a \neq 0$

Universal families

Integer numbers

Let integer a, b be sampled uniformly from $[1; p)$ and $[0; p)$.

Let's analyze probability of collision:

$$h_{a,b}(x) = \big((ax + b) \bmod p\big) \bmod M$$

$$h_{a,b}(x) = h_{a,b}(y)\,, \qquad x \neq y$$

$$ax + b \equiv ay + b + iM \pmod{p} \qquad \mathrm{i} \in \left[0; \left\lfloor \frac{p-1}{M} \right\rfloor\right]$$

$$a(x - y) \equiv iM \pmod{p}$$

p − prime
$x \neq y$

$$\boldsymbol{a \equiv iM\,(x - y)^{-1} \pmod{p}}$$

For each pair (x, y) amount of a which create collision: $\left\lfloor \frac{p-1}{M} \right\rfloor$

Total amount of a: $p - 1$.

$$P[C_h(x, y) = 1] = \left\lfloor \frac{p-1}{M} \right\rfloor / (p - 1) \leq \frac{p-1}{M} / (p - 1) = \frac{1}{M} \quad \blacksquare$$

Universal families

Vectors, strings

Vectors:

$$\overline{x} = (x_0, x_1, \dots, x_{k-1})$$

$$H = \{(h_0, h_1, \dots h_{k-1}) \colon h_i \text{ is sampled independently from } H \text{ for } x_i\}$$

$$h(\overline{x}) = \sum_{i=0}^{k-1} h_i(x_i) \;\; mod\ M$$

Strings: Polynomial hash.

$$s = (s_0 s_1 \dots s_l)$$

$$s_i \in [1; |\Sigma|], \qquad p\colon \text{prime} \geq \max(|\Sigma|, M), \qquad a \sim U([1; p))$$

$$h_a = h_{int}\left(\sum_{i=0}^{l} s_i\, a^i \;\; mod\ p\right)$$

, where h_{int} – universal mapping $[0, p) \to [0, M)$

$$P[C(x, y) = 1] \leq \frac{l}{M}$$

Rabin-Karp algorithm

- **Idea**
- Rolling hash
- Implementation

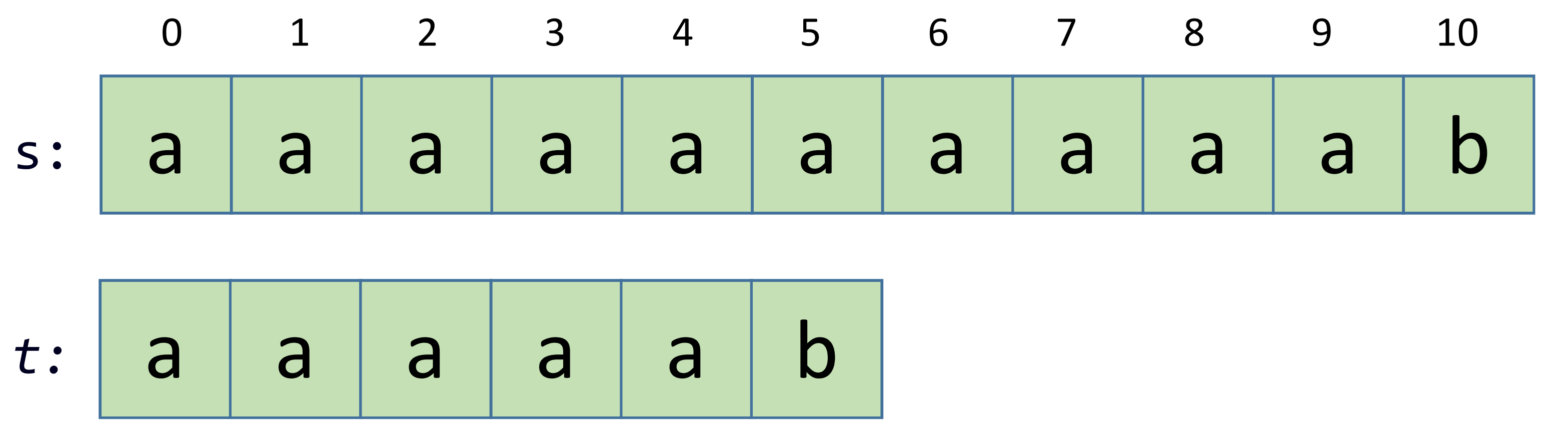
Rabin-Karp algorithm

Idea

Rabin-Karp algorithm solves string-searching problem in linear time $O(|s| + |t|)$ (like KMP).

Let’s remember naïve solution of string-searching problem:

```
substrings = []
for i in range(len(s) - len(t) + 1):
    if all([s[i + j] == t[j] for j in range(len(t))]):
        substrings.append(i)
```



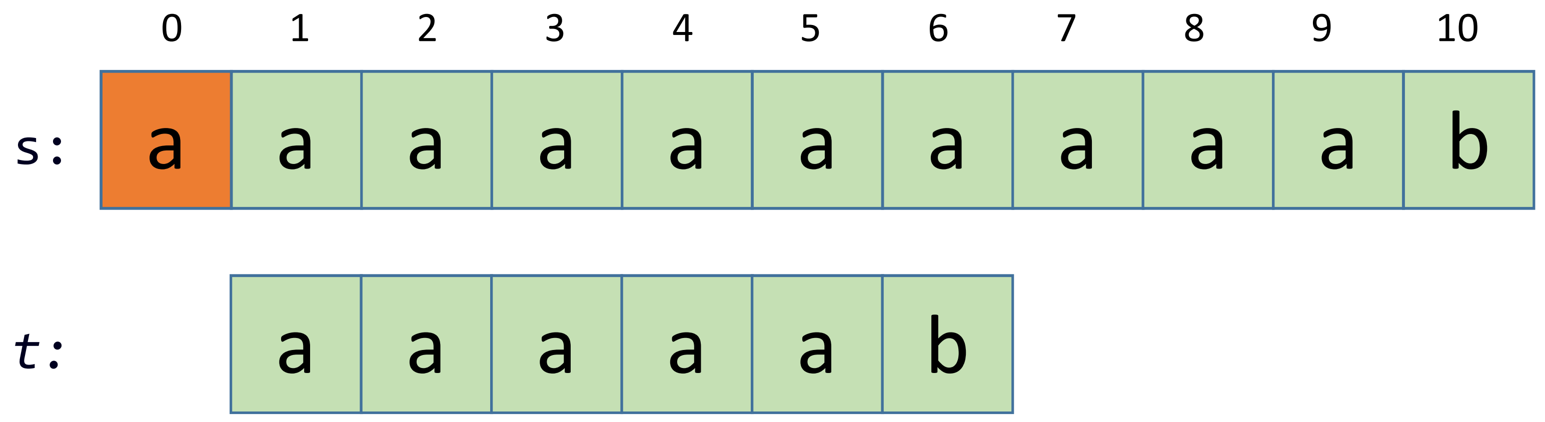
Rabin-Karp algorithm

Idea

Rabin-Karp algorithm solves string-searching problem in linear time $O(|s| + |p|)$ (like KMP).

Let's remember naïve solution of string-searching problem:

```
substrings = []
for i in range(len(s) - len(t) + 1):
    if all([s[i + j] == t[j] for j in range(len(t))]):
        substrings.append(i)
```



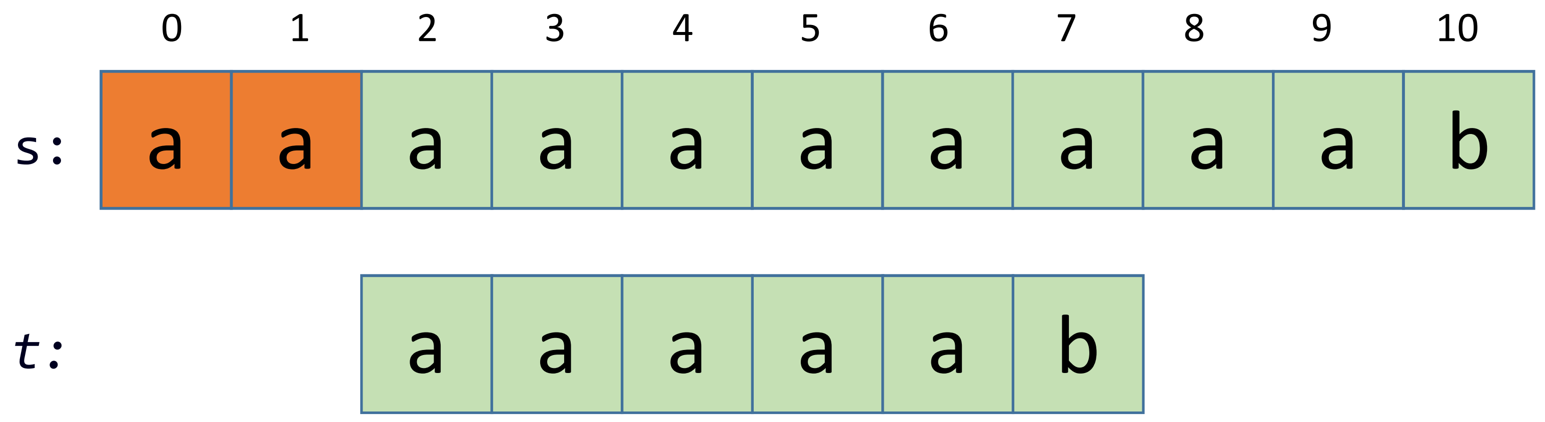
Rabin-Karp algorithm

Idea

Rabin-Karp algorithm solves string-searching problem in linear time $O(|s| + |p|)$ (like KMP).

Let’s remember naïve solution of string-searching problem:

```
substrings = []
for i in range(len(s) - len(t) + 1):
    if all([s[i + j] == t[j] for j in range(len(t))]):
        substrings.append(i)
```



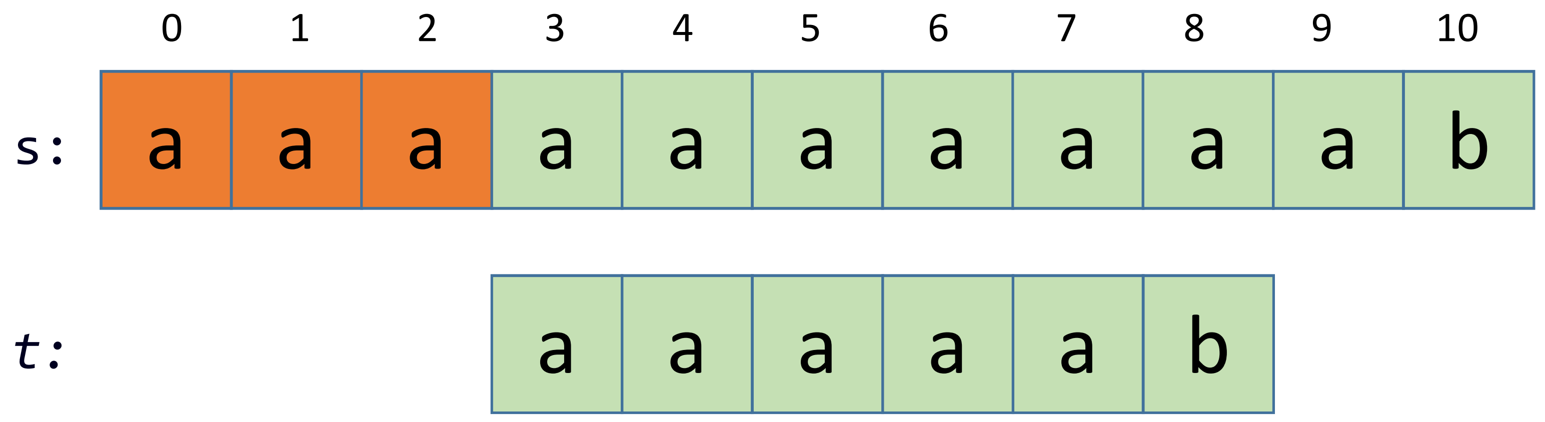
Rabin-Karp algorithm

Idea

Rabin-Karp algorithm solves string-searching problem in linear time $O(|s| + |p|)$ (like KMP).

Let’s remember naïve solution of string-searching problem:

```
substrings = []
for i in range(len(s) - len(t) + 1):
    if all([s[i + j] == t[j] for j in range(len(t))]):
        substrings.append(i)
```



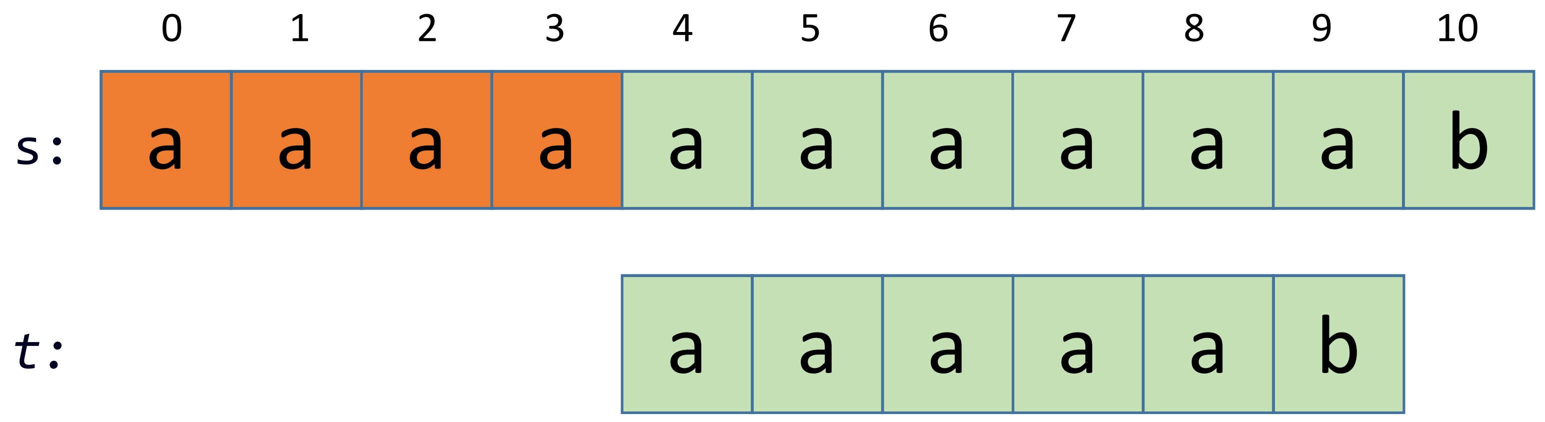
Rabin-Karp algorithm

Idea

Rabin-Karp algorithm solves string-searching problem in linear time $O(|s| + |p|)$ (like KMP).

Let's remember naïve solution of string-searching problem:

```
substrings = []
for i in range(len(s) - len(t) + 1):
    if all([s[i + j] == t[j] for j in range(len(t))]):
        substrings.append(i)
```



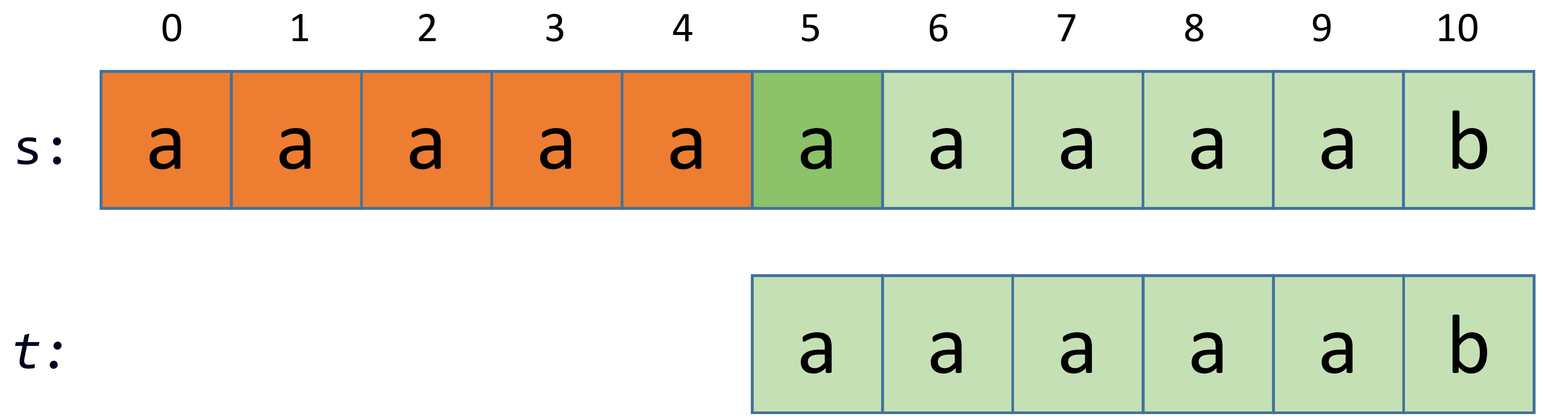
Rabin-Karp algorithm

Idea

Rabin-Karp algorithm solves string-searching problem in linear time $O(|s| + |p|)$ (like KMP).

Let’s remember naïve solution of string-searching problem:

```
substrings = []
for i in range(len(s) - len(t) + 1):
    if all([s[i + j] == t[j] for j in range(len(t))]):
        substrings.append(i)
```



Rabin-Karp algorithm

Idea

The main problem in naïve solution is string comparison which takes $O(|t|)$ operations.

Let’s use hashes to speed up string comparison.

But to do that we need to calculate hashes on all substring of length $|t|$. Naïve calculation will take the same $O(|s||t|)$.

But a simple trick with polynomial hash allows to reuse hashes for already calculated parts of string.

Rabin-Karp algorithm

- Idea
- **Rolling hash**
- Implementation

Rabin-Karp algorithm

Rolling hash

Let’s use polynomial hash:

$$hash = \left(\sum_{i=0}^{l} s_i\, a^i\right) mod\ p$$

$P(collision) \sim \frac{1}{p}$

So, p should be big prime

e.g.: $10^9 + 7; 10^9 + 9$

$$p - \text{prime} \geq |\Sigma|, \qquad a \in [1; p)$$

Should be random
to provide universality.
Otherwise, exists input which
drops complexity to $O(|s||p|)$
In practice, you’d better
choose $a \geq |\Sigma|$

Let’s calculate hash for the pattern: $h_t = hash(t)$

Structure of polynomial hash allows to use recurrent formula:

$$hash(s_0 s_1 \ldots s_{k-1} s_k) = \left(hash(s_0 s_1 \ldots s_{k-1}) + s_k a^k\right) mod\ p$$

Rabin-Karp algorithm

Rolling hash

$$hash(s_0 s_1 \dots s_{k-1} s_k) = \left(hash(s_0 s_1 \dots s_{k-1}) + s_k a^k\right) \, mod \, p$$

Let’s use idea similar to prefix-sum. Let’s calculate prefix-hash using this recurrent formula:

$$h[i] = hash(s[:i])$$

$$h[0] = 0, \qquad h[i+1] = \left(h[i] + s[i] * a^i\right) \, mod \, p$$

```
h = [0] * (len(s) + 1)
a_pow = [1] * (len(s) + 1)
for i in range(len(s)):
    h[i + 1] = (h[i] + s[i] * a_pow[i]) % p
    a_pow[i + 1] = (a_pow[i] * a) % p
```

Now, we have hashes for all prefixes of s. Let’s use it for comparison in naïve string-searching solution.

Rabin-Karp algorithm

Rolling hash

$$h[i] = hash(s[:i])$$

Similarly to prefix-sum approach, we can try to calculate hash on a range $[l; r)$:

$$h[r] - h[l] = hash(s[:r]) - hash(s[:l]) =$$

$$\left(\sum_{i=0}^{r-1} s_i\, a^i\right) mod\ p - \left(\sum_{i=0}^{l-1} s_i\, a^i\right) mod\ p = \left(\sum_{i=l}^{r-1} s_i\, a^i\right) mod\ p$$

But,

$$hash(s[l:r]) = \left(\sum_{i=l}^{r-1} s_i\, a^{i-\boldsymbol{l}}\right) mod\ p$$

$$h[r] - h[l] = hash(s[l:r]) * a^l$$

Rabin-Karp algorithm

Rolling hash

To perform our algorithm we need to compare h_t with hash on each substring: $hash(s[i: i + |t|])$.

$$h_t \quad \overset{?}{\equiv} \quad hash(s[i: i + |t|]) \quad (mod\ p)$$

But we can easily calculate $hash(s[i: i + |t|]) * a^i = h[i + |t|] - h[i]$.

As p is prime, we can calculate a^{-i} in the field, but it’s not necessary.

We can just compare:

$$h_t * a^i \overset{?}{\equiv} hash(s[i: i + |t|]) * a^i \qquad (mod\ p)$$

$$h_t * a^i \overset{?}{\equiv} h[i + |t|] - h[i] \qquad (mod\ p)$$

Equality means that either strings are equal, or collision appeared.
We should compare strings explicitly in this case.
That leads us to Rabin-Karp algorithm.

Rabin-Karp algorithm

- Idea
- Rolling hash
- **Implementation**

Rabin-Karp algorithm

Implementation

```python
def find_substrings_rabin_karp(s, t):
    h = [0] * (len(s) + 1)
    a_pow = [1] * (len(s) + 1)
    for i in range(len(s)):
        h[i + 1] = (h[i] + c_to_i(s[i]) * a_pow[i]) % p
        a_pow[i + 1] = (a_pow[i] * a) % p
    h_t = 0
    for i in range(len(t)):
        h_t = (h_t + c_to_i(t[i]) * a_pow[i]) % p
    substrings = []
    for i in range(len(s) - len(t) + 1):
        if (h_t * a_pow[i]) % p == (h[i + len(t)] - h[i]) % p:
            if all([s[i + j] == t[j] for j in range(len(t))]):
                substrings.append(i)
    return substrings
```

$$O\big(|s| + |t| + |t| * (n_{matches} + n_{collisions})\big)$$

$$n_{collisions} \sim P(collision) * n_{comparisons} \le \frac{|t|}{p} * (|s| - |t|) = O\left(\frac{|s||t|}{p}\right) \xrightarrow{p\, =\, O(|s||t|)} O(1)$$

$$O\big(|s| + |t|(n_{matches} + 1)\big)$$

Hash table

- **Idea**
- Separate chaining collision handling (Linked list)
- Open addressing collision handling

Hash table

Idea

Let's suppose that we need to implement a data structure (associative array) which supports the following operations:

`insert(k, v)` – assign value v to key k (or update)

`get(k)` – get last value assigned to key k

`remove(k)` – remove value assigned to key k

This data structure depends on set of possible keys K.

Hash table

Idea

E.g. if $K = \{0, \dots, k_{max}\}$, we can just use vector (list) and use key as index in this vector:

```
insert(k, v): data[k] = v
get(k): return data[k]
remove(k): data[k] = None
```

For more complex keys (e.g. K – set of strings) is will not work.

But we can use hash $h: K \rightarrow V = \{0, \dots, M-1\}$ to convert our keys to suitable range and use obtained hash values as indices of vector (list).

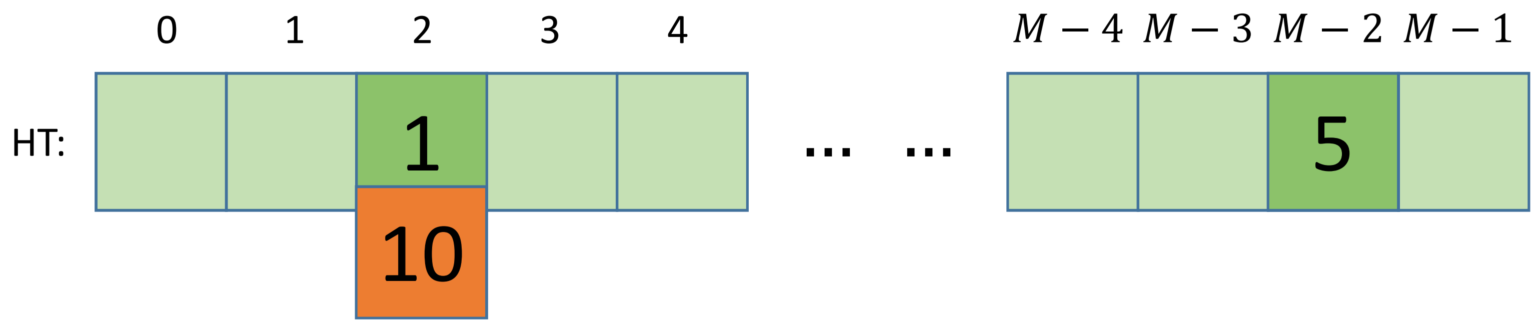
Hash table

Idea

insert(k_1, 1) $\rightarrow$ $h(k_1) = 2$

insert(k_2, 5) $\rightarrow$ $h(k_2) = M - 2$

insert(k_3, 10) $\rightarrow$ $h(k_3) = 2$



Collisions should be resolved!
Otherwise structure may return wrong answer.

Hash table

- Idea
- **Separate chaining collision handling (Linked list)**
- Open addressing collision handling

Hash table. Separate chaining.

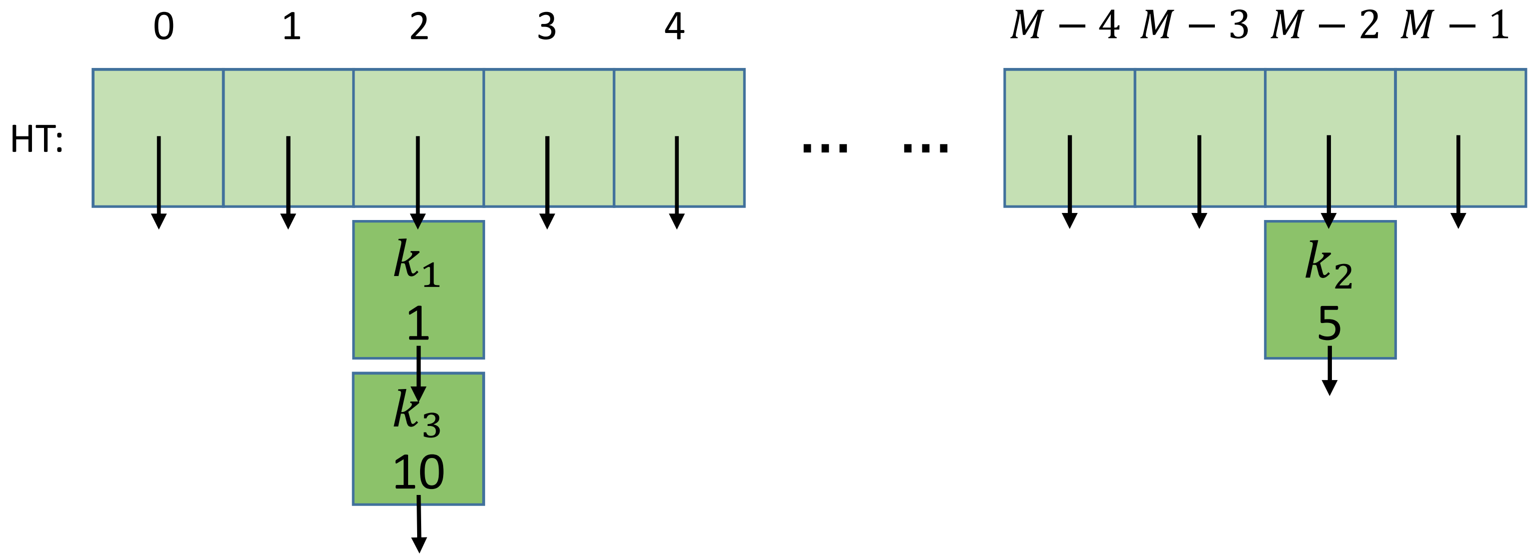
Idea

Let’s store linked list in each item of hash table.

insert(k_1, 1) $\rightarrow$ $h(k_1) = 2$

insert(k_2, 5) $\rightarrow$ $h(k_2) = M - 2$

insert(k_3, 10) $\rightarrow$ $h(k_3) = 2$



Hash table. Separate chaining.

Idea

So, to perform **get($\boldsymbol{k}$)** we should:

Iterate over linked list $HT[h(\mathrm{k})]$ and return one with key = k if any.

To perform **remove($\boldsymbol{k}$)** we should:

Iterate over linked list $HT[h(\mathrm{k})]$ and remove node with key = k if any.

To perform **insert($\boldsymbol{k}$)** we should:

Iterate over linked list $HT[h(k)]$ and update node with key = k if any, otherwise add new node.

In all cases we should iterate over linked list $HT[h(k)]$. So, number of operations equals length of this list. Let's estimate expectation of length of this list in hash table with N items in it.

Hash table. Separate chaining.

Complexity

Let’s suppose, our structure contains values for N keys. Set of these keys: A.

Let’s calculate expectation of length of chain for $k \in A$: L_s.

$$L_k = \sum_{k' \in A} [h(k) = h(k')] = \sum_{k' \in A} C_h(k, k') + 1$$

$$E(L_k) = E\left(\sum_{k' \in A} C_h(k, k') + 1\right) = P(collision) * (N - 1) + 1 \le \frac{N}{M} + 1$$

$$E(L_k) = O(\alpha + 1), \qquad where\ \alpha = \frac{N}{M}\text{: load factor}$$

This means that computational complexity of all operations for hash table is:

$$O(\alpha + 1)$$

When we add too much items to hash table and α becomes big, we can increase M and perform rehashing. This operation will take $O(N)$ instructions, but similarly to reallocation in vector, amortized complexity will be $O(1)$.

So, we can keep $\alpha = O(1)$. This makes the complexity of each operation to be $O(1)$.

Hash table

- Idea
- Separate chaining collision handling (Linked list)
- **Open addressing collision handling**

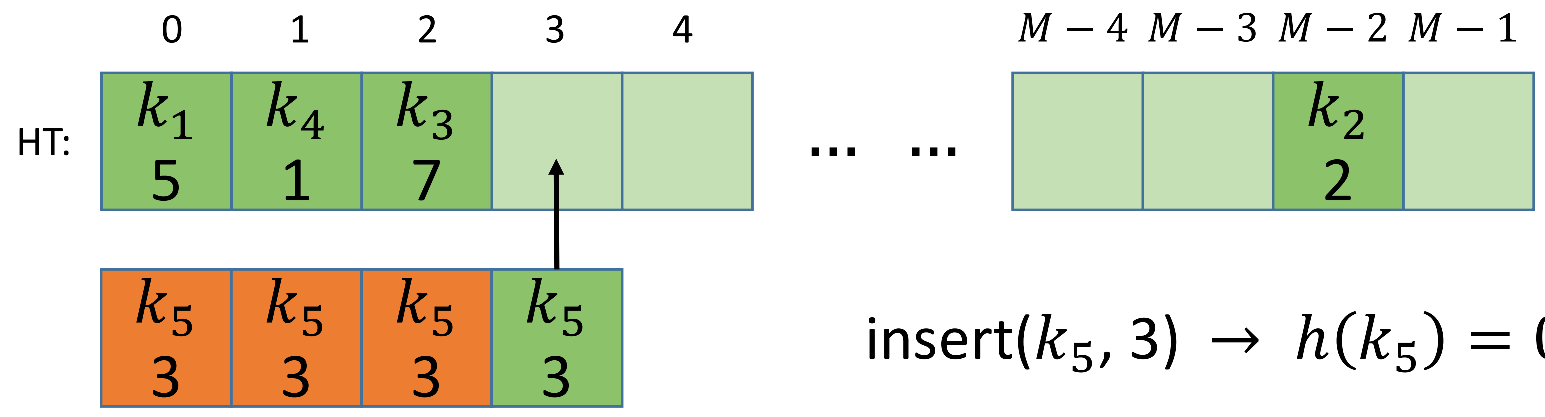
Hash table. Open addressing.

Linear probing

There is another idea for collision handling, which does not include linked lists.

The idea is the following: let's store values and keys directly in HT. If collision appears, let's move to the right until we find a free place in HT.

```
i = h(k)
while HT[i] is not None:
    i = (i + 1) % p
```



insert(k_5, 3) $\rightarrow$ $h(k_5) = 0$

Hash table. Open addressing.

Double hashing

What if we have several items with close hashes. In this case, we can obtain long chain.

There is a modification which makes size of a step different for different objects. It is called double hashing. Two hashes are defined:

h_1 – defines initial position in HT.

h_2 – defines size of step.

```
i = h1(k)
while HT[i] is not None:
    i = (i + h2(k)) % p
```

Hash table. Open addressing.

Load factor

So, open addressing uses HT array to store keys and values. If α is small, it works faster than chaining approach, but when α becomes close to 1, number of operations we perform to find a place increases dramatically.

So, to provide better performance, we should maintain α rather small.

Conclusion

Python built-ins

Python has built-in functionality for hashing:

```
hash(x)
```

For integer numbers hash is determined. For positive ones:

```
hash(x) = (x % (2**31-1))
```

And that leads to possible collisions.

But for strings, hash functions are sampled from universal family. So, hash values are undetermined for different launches. You may try to launch this command several times:

```
> python -c print(hash('a'))
```

Python built-ins

You may implement hashing procedure for your class by defining `__hash__` method:

```python
class Hashable:
    def __init__(self, name):
        self.name = name

    def __hash__(self):
        return hash(self.name)
```

It is supposed that:

1. hash do not change while instance exists.
2. $a == b \Rightarrow hash(a) == hash(b)$

Python built-ins

Hash tables are already implemented in python.

They are backends of **dict** and **set** containers.

Hash tables in **dict** and **set** implementation use open addressing with double hashing and α is kept $\leq \frac{2}{3}$.

You can read more about it in the comments in the source code:

https://github.com/python/cpython/blob/master/Objects/dictobject.c#L135

Thank you for watching!