Hashing and hash tables

1 Hashing

Hashing if a procedure of mapping data of arbitrary size to fixed-size values. Function that implements this procedure is called a **hash function**, its result is **hash**. Hash function:

- gives an equal hash for the equal data;
- *sometimes* gives an equal hash for the unequal data.

TThe second property is the result of the hashes being a fixed size while input data is unlimited. As a result, we get a function from a set of larger size to a set of smaller size. So, for a single hash value, there are multiple preimages from a set of input data by pigeonhole principle (or Dirichlet's box principle). This problem is called a **collision**. If there is a limited set of input data in some problem, we can find a hash function, that will be an injective function (**perfect hash function**). However, in general, perfect hashing is impossible. So, the collision probability shows the quality of hash function.

A hash function has the following properties:

- Uniformity – the results of the hash function must be uniformly distributed over its values range; if values are stacked in one part, the collision probability increases;
- Efficiency;
- Deterministic – it's good when the function doesn't depend on random values; hash of the same data mustn't change between program executions;
- Defined range – you always know how much memory you need to allocate; иusing hashes of undefined size is possible but not desired.

1.1 Polynomial hash

Let's implement some simple but effective algorithm. We define hash function as

$$h(s) = \sum_{i=0}^{N} b^{N-i} \cdot code(s_i) \tag{1}$$

or

$$h(pref_i) = b \cdot h(pref_{i-1}) + code(s_{i-1}), \tag{2}$$

where $N = len(s) - 1$, $pref_i$ - prefix of length i, b - base, $code(s_i)$ - symbol code. Formula 1 gives polynomial of degree $N - 1$. Formula 2 defines hash function as a recursive function, that we'll use.

You should pay some attention to choosing a base and code function, also base depends on codes. As a code, you can use the ASCII table or alphabetical number. For ex For example, if the problem's statement guarantees that strings contain only small Latin letters, alphabetical numbers will be a good choice. But there is one thing: немук use 0 as a symbol code. Let $code("a") = 0$, then

$$code("a") = code("aa") = ... = 0 \tag{3}$$

The base must exceed the maximum code value to avoid additional collisions.

However, it is worth noting that we do not limit the polynomial hash to anything, which allows us to avoid collisions. But the hash is growing pretty fast. In this case, there are two possible solutions: use modulo division or use long arithmetic.

The first option is widely used in languages where there is no built-in long arithmetic. Moreover, the integer data type where the hash is stored automatically performs this division (in a result of type overflow extra bits are automatically lost). However, the usage of type overflow is a bad idea. Because sizes of data types are powers of two, we can say that a polynomial hash is calculated modulo $M = 2^k$. But in this case, you can easily pick up the data, which will produce a large number of collisions (see 1.2). Therefore you should use some other value for M. A good solution would be to use a simple M. $2^{31} - 1$ for 32-bit machines and $2^{61} - 1$ for 64-bit machines are especially good because modulo division for them can be replaced by bitwise operations.

The second option has a lower chance of collisions (we still cannot support infinitely large numbers and they will still be limited). However, support for more hash values will cost additional memory and the time it takes to compare two hashes.

We will write both variants of polynomial hashes. Python has already implemented long arithmetic, so we will use it. For example, we assume that the string consists of lowercase Latin letters. The base will take the number 37.

Listing 1: "Unlimited" polynomial hash

```
B = 37

def poly_hash(s):
    h = 0
    for c in s:
        h = h * B + ord(c) - ord('a') + 1
    return h
```

Listing 2: Polynomial hash with modulo

```
B = 37
# Subtraction has more priority than `shift left`
M = (1 << 61) - 1

def poly_hash(s):
    h = 0
    for c in s:
        h = (h * B + ord(c) - ord('a') + 1) % M
    return h
```

1.2 Anti-hash

As mentioned earlier, when writing polynomial hashes with division modulo probability of collisions increases. Usually, people use type overflow instead of explicit division. For instance, using the `unsigned long long int` type in C ++, in fact, a hash is calculated modulo 2^{64}. In the case of the module of degree two that there arises the possibility of selecting an anti-hash string in which a large number of collisions occur. The string has the form `ABBABAABBAABABBABAABABBAABBAB...` and is called the Thue-Morse sequence. The participant of competitions nicknamed Zlobober wrote about this fact on his blog on the website codeforces.com. A good solution to this problem is not to use overflows (solution using long arithmetic or division modulo of some other number).

2 Rabin-Karp algorithm

The algorithm allows you to search for occurrences of the string S in the text T using hashes. It consists of the following: count the hashes for all the text prefixes of T, iterate over all the substrings of length $|S|$ in T and compare with S. Let us estimate the running time of such an algorithm. The time it takes to hash a string is $O(|S|)$. Recall that hash counting for a string was expressed recursively through a hash of a string shorter length. Thus, counting the hash for the string, we simultaneously get hashes for all of its prefixes, which means that for the text T we need $O(|T|)$ time. Iterate over all the substrings fixed-length requires a linear amount of time on the length of the string T. Comparison of two lines we will execute for $O(1)$ thanks to hashes. Total asymptotic of the algorithm is $O(|S| + |T|)$.

To implement the algorithm, we change the hash function. To reduce the code, we consider that all constants are already declared. For example, a solution with a limited hash is given.

```
    def poly_hash(s):
    h = [0]  # Array of hashes for all prefixes
    for c in s:
```

```
        h.append((h[-1] * B + ord(c) - ord('a') + 1) % M)
    return h
```

Now we store hashes for all prefixes. For conveniece we store hash for empty prefix in $h[0]$.

Now we need to calculate the hash for arbitrary substring. Suppose we want to compute a hash of substring $[l, r)$. Note that the hash of our substring will differ from the hash of the prefix of length r in the first in the first l members of the polynomial. $h[l]$ just will contain only these first l members, but with lesser exponent. Multiplying $h[l]$ by B^{r-l}, we get the same l members of the polynomial with the desired exponent. Then the hash of the substring is calculated as $h[r] - h[l] * B^{r-l}$. In our case $r - l$ will always be the same as S doesn't change, so $r-l = |S|$. Then we can calculate $B^|S|$ once and use it for computing hashes of substrings.

3 Hash table

Associative array is an abstract data structure composed of collection of **key/value** pairs, such that each possible key appears at most once in the collection. Operations:

- add a new pair
- remove pair
- search by key

A **hash table** is one of the possible implementations.

The operation in the hash table begins with the calculation of the hash function of the key. The resulting hash value $i = hash(key)$ plays the role of an index in the H array. Then the operation performed (add, delete or search) is redirected to an object that is stored in the corresponding cell of the array, in $H[i]$.

However, do not forget about the collisions. Such events are not so rare - for example when inserted into a hash table of 365 cells only 23 elements collision probability will already exceed 50% (if each element can be put into any cell with equal probability) - see the birthdays problem.

When working with a hash table, the **loading factor** (a ratio of number of entries to the size of the array H). With the growth of the coefficient performance decreases (time spent on collisions resolution). Upon reaching a certain level of laoding, you can increase the size of the table, which will reduce the loading factor. In the process of expanding the table stored items are rehashed. However, even after expansion, collisions are still possible. Therefore, the mechanism for collisions resolution is an important component of any hash table.

There are two main mechanisms: separate chaining and open addressing. The hash table contains some array H, whose elements are pairs (hash table

with open addressing) or lists of pairs (hash table with separate chaining). We will omit the method of open addressing here.

Let's look at a hash table with separate chaining. Each element of the H will be a linked list of pairs (key, value) corresponding to the same hash of the key. Collisions are just lead to the appearance of chains longer than one element. Search or delete operations elements require viewing all the elements of the corresponding chain in order to find an element with given key. To add an item, you need to add the item to the end or beginning of the corresponding list.

An important property of hash tables is that, with some reasonable assumptions, all three operations (search, insert, delete items) are performed on average for $O(1)$. But it is not guaranteed that the execution time of a single operation is short.

3.1 Implementation

Let's write a hash table that will implement a set. The set is also associative array, but instead of pairs (key, value) stores only keys (we can assume that all values are None). We will start an array whose size will be a reasonably large number (for example, $10^5 + 7$). Each element of the array will be initialized with an empty list. Define our hash function as follows: the remainder of dividing the polynomial hash for our key by the length of our array.

```
TABLE_SIZE = 10 ** 5 + 7


def get_pos(key):
    h = poly_hash(key)
    return (h % TABLE_SIZE, h)
```

The first number returned by the function will determine the cell where the pair is to be stored. The second number is a polynomial hash. We will store it in the table along with the key, which will allow us to compare keys quickly when resolving collisions. Defining the cell of the table where it should be stored element, we check the list that the cell points to. The check is to compare the key of the item to be added/searched/deleted with the keys of the list items (here we will compare hashes instead of the keys themselves).

```
def add(table, key):
    pos, h = get_pos(key)
    for e in table[pos]:
        if e[0] == h:
            print(key, "is already in table")
            return
    table[pos].append([h, key])
```

```
def find(table, key):
    """
    Returns:
        True, if key is in table.
        False, otherwise.
    """
    pos, h = get_pos(key)
    for e in table[pos]:
        if e[0] == h:
            return True
    return False


def delete(table, key):
    pos, h = get_pos(key)
    for i, e in enumerate(table[pos]):
        if e[0] == h:
            table[pos].pop(i)
            return
    print(key, "is not in table")
```

Creating an empty table at the beginning of the program:

```
table = [[] for _ in range(TABLE_SIZE)]
```