Хеширование и хеш-таблицы

1 Хеширование

Хеширование - процесс сопоставления значений фиксированного размера данным произвольной длины, выполняемое определенным алгоритмом. Функция, воплощающая такой алгоритм называется **хеш-функцией**, а результат преобразования - **хешем** или **хеш-суммой**. Хеш-функция ведет себя следующим образом:

- одинаковые данные всегда дают одинаковый хеш;
- разные данные *иногда* дают одинаковый хеш.

Второе свойство возникает из-за того, что хеши имеют установленный размер, в то время как входные данные этим не ограничены. В результате мы получаем, что хеш-функция делает отображение из ножества входных данных во множество хешей, мощность которого будет сильно меньше. По принципу Дирихле на один хеш будет приходиться несколько различных входных данных. Такие совпадения называются **коллизией**. Если в какой-либо задаче входные данные ограничены, можно подобрать такое множество хешей, что его мощность будет превышать мощность множества входных данных. В таком случае мы можем построить хеш-функцию, задающую инъективное отображение (**идеальное хеширование**). Однако, в общем случае возникновение коллизий неизбежно. Вероятность возникновения коллизий используется для оценки качества хеш-функции.

Вообще хеш функция обладает следующими свойствами:

- Равномерность – результаты хеш функции должны быть равномерно распределены по всему интервалу значений; если данные будут кучковаться в какой-то одной части, то вероятность коллизий возрастет;
- Эффективность;
- Детерминированность – хорошо, когда функция не зависит от каких либо случайных чисел; хеш для одних и тех же данных не должен меняться от запуска к запуску программы;
- Фиксированный интервал – под такие хеши удобно заготавливать нужный объем памяти, всегда известен его размер; использование хешей нефиксированного интервала возможно, но менее приветствуется.

1.1 Полиномиальный хеш

Рассмотрим простой, но эффективный алгоритм хеширования. Определим нашу хеш-функцию следующим образом:

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

или

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

где $N = len(s) - 1$, $pref_i$ - префикс длины i, b - основание, $code(s_i)$ - код символа. Если развернуть формулу 1, то получится полином порядка N (отсюда и название). Формула 2 задает хеш в рекурсивном виде и будет использована при написании кода.

Стоит обратить внимание на код символа и основание, ибо от кодов будет зависеть выбор основания. Кодом может быть как код символа в таблице ASCII, так и просто порядковый номер в алфавите. Например, если в задаче гарантируется, что любая строка состоит только из строчных букв английского алфавита, то порядковый номер - хороший вариант для кодов символов. Самое главное, не используйте код 0 для какого либо символа. Например, $code("a") = 0$, тогда

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

Основание должно превышать максимальный код какого-либо из возможных символов в строке, это позволит избежать дополнительных коллизий.

Однако, стоит заметить, что мы ни чем не ограничиваем полиномиальный хеш, что позволяет избегать коллизий. Но хеш растет довольно быстро. В таком случае есть два возможных выхода: использовать операцию деления по модулю или использовать длинную арифметику.

Первый вариант широко используется в языках, где нет встроенной длинной арифметики. Более того, целочисленный тип данных, где хранится хеш, автоматически выполняет это деление (в результате переполнения типа лишние биты автоматически теряются). Однако, использовать переполнение типов является плохой идеей. Т.к. размеры типов данных являются степенями двойки, то можно считать, что полиномильный хеш считается по модулю $M = 2^k$. Но в этом случае можно легко подобрать строки, которые будут давать большое колличество коллизий (см. 1.2). Поэтому стоит использовать какое-нибудь свое значение M. Хорошим решением будет использовать простые M. Особенно хороши $2^{31} - 1$ для 32-битных машин и $2^{61} - 1$ для 64-битных машин, т.к. для них деление по модулю можно заменить побитовыми операциями.

Второй вариант имеет меньший шанс возникновения коллизий (мы все равно не можем поддерживать бесконечно большие числа и их придется ограничить). Однако поддержка большего множества хешей обойдется дополнительной памятью и временем, которое необходимо на сравнение двух хешей.

Напишем оба варианта полиномиальных хешей. В языке Python уже реализована длинная арифметика, чем мы и воспользуемся. Для примера будем считать, что строка состоит из строчных английских букв. Основанием возьмем число 37.

Listing 1: «Неограниченный» полиномиальный хеш

```
B = 37

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

Listing 2: Полиномиальный хеш по модулю

```
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 Антихеш

Как упоминалось ранее, при написании полиномиальных хешей с делением по модулю возникает вероятность коллизий. Обычно пользуются переполнением типа вместо явного деления. Например, используя в C++ тип `unsigned long long int`, по факту считают хеш по модулю 2^{64}. Именно в случае модуля степени двойки и возникает возможность подбора строки-антихеша, в которой возникает большое количество коллизий. Строка имеет вид `ABBABAABBAABABBABAABABBAABBAB...` и называется строкой Туэ-Морса. Про данный факт подробно написал участник соревнований на сайте codeforces.com под ником Zlobober в своем блоге. Хорошим решением этой проблемы является не использование переполнений (решение при помощи длинной арифметики или использование деления по модулю другово числа).

2 Алгоритм Рабина-Карпа

Алгоритм позволяет искать вхождения строки S в текст T при помощи хешей. Заключается он в следующем: считаем хеши для всех префик-

сов текста T, перебираем все подстроки длины $|S|$ в T и сравниваем с S. Оценим время работы такого алгоритма. Время, необходимое для хеширования строки составляет $O(|S|)$. Вспомним, что подсчет хеша для строки выражался рекурсивно через хеш строки меньшей длины. Таким образом, считая хеш для строки, мы попутно получаем хеши для всех ее префиксов, а значит для строки T нам надо $O(|T|)$ времени. Перебор всех подстрок фиксированной длины требует линейное количество времени от длины строки T. Сравнение двух строк мы будем выполнять за $O(1)$ благодаря хешам. Итого асимптотика алгоритма - $O(|S| + |T|)$.

Для реализации алгоритма изменим функцию вычисления хеша. Для сокращения вставок с кодом, считаем что все константы уже объявлены. Для примера приводится решение с ограниченным по модулю хешом.

```
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
```

Теперь мы храним хеши для всех префиксов. Для удобства при написании кода в нулевой ячейке массива мы храним хеш для префикса длины 0.

Теперь научимся вычислять хеш для произвольной подстроки. Пусть мы хотим вычислить хеш подстроки $[l, r)$. Заметим, что хеш нашей подстроки будет отличаться от хеша префикса длины r на первые l членов полинома. $h[l]$ как раз и будет содержать в себе только эти первые l членов, но меньшей степени. Домножив $h[l]$ на B^{r-l}, мы получим те самые l членов полинома с нужной нам степенью. Тогда хеш подстроки вычисляется как $h[r] - h[l] * B^{r-l}$. В нашем случае $r - l$ не меняется, т.к. S не меняется, и $r - l = |S|$. Тогда вычислим $B^{|}S|$ единожды и будет использовать при подсчете хешей подстрок.

3 Хеш-таблица

Ассоциативный массив - абстрактная структура данных, которая хранит пары (ключ, значение), при том каждый ключ встречается не более одного раза. Такой массив поддерживает три типа операций:

1. добавление новой пары;
2. поиск по ключу;
3. удаление пары по ключу.

Одной из реализаций ассоциативного массива является **Хеш-таблица**.

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение $i = hash(key)$ играет роль

индекса в массиве H. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива, в $H[i]$.

Однако, не стоит забывать про коллизии. Такие события не так уж и редки — например, при вставке в хеш-таблицу размером 365 ячеек всего лишь 23-х элементов вероятность коллизии уже превысит 50% (если каждый элемент может равновероятно попасть в любую ячейку) — см. парадокс дней рождения.

При работе с хеш-таблицей вводят величину **коэффициент заполнения** (кол-во хранимых элементов, делённое на размер массива H). С ростом коэффициента подает производительность (затраты времени на разрешение коллизий). По достижению определенного уровня заполнения можно прибегнуть к увеличению таблицы, что снизит коэффициент заполнения. В процессе расширения таблицы хранимые элементы перехешируются. Однако даже после расширения все равно возможны коллизии. Поэтому механизм разрешения коллизий — важная составляющая любой хеш-таблицы.

Существуют два основных механизма: с цепочками и открытой адресацией. Хеш-таблица содержит некоторый массив H, элементы которого есть пары (хеш-таблица с открытой адресацией) или списки пар (хеш-таблица с цепочками). Способ открытой адресации тут мы рассматривать не будем.

Рассмотрим хеш-таблицу с цепочками. Каждый элемент массива H будет представлять собой связный список пар (ключ, значение), соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента. Операции поиска или удаления элемента требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. Для добавления элемента нужно добавить элемент в конец или начало соответствующего списка.

Важное свойство хеш-таблиц состоит в том, что, при некоторых разумных допущениях, все три операции (поиск, вставка, удаление элементов) в среднем выполняются за время $O(1)$. Но при этом не гарантируется, что время выполнения отдельной операции мало.

3.1 Реализация

Напишем хеш-таблицу, которая будет реализовывать множество. Множество также является ассоциативным массивом, но вместо пар (ключ, значение) хранит только ключи (можно считать, что все значения None). Заведем массив, размер которого будет являться разумно большим числом (например, $10^5 + 7$). Каждый элемент массива будет проинициализирован пустым списком. Определим нашу хеш-функцию следующим образом: остаток от деления полиномиального хеша для нашего ключа на длину нашего массива.

```
TABLE_SIZE = 10 ** 5 + 7


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

Первое число, возвращенное функцией, будет определять ячейку, где должна храниться пара. Второе число - полиномиальный хеш. Его мы будем хранить в таблице вместе с ключом, что позволит нам сравнивать ключи быстро при разрешении коллизий. Определив ячейку таблицы, где должен храниться элемент, мы проверяем список, на который указывает ячейка. Проверка заключается в сравнении ключа добавляемого/искомого/удаляемого элемента с ключами элементов списка (вот тут мы и будем сравнивать хеши вместо самих ключей).

```
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")
```

Создание пустой таблицы в начале программы:

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