Хэширование в строковых задачах

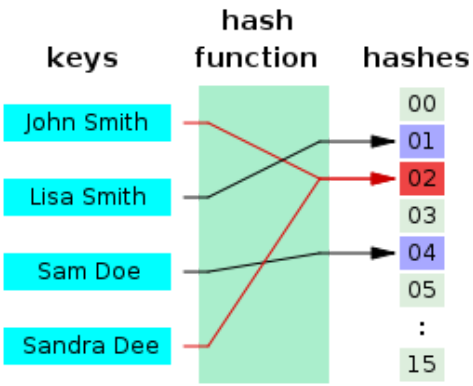
Хэш — это какая-то функция, сопоставляющая объектам какого-то множества числовые значения из ограниченного промежутка.

«Хорошая» хэш-функция:

- Быстро считается — за линейное от размера объекта время;
- Имеет не очень большие значения — влезающие в 64 бита;
- «Детерминированно-случайная» — если хэш может принимать n различных значений, то вероятность того, что хэши от двух случайных объектов совпадут, равна примерно $\frac{1}{n}$.

Обычно хэш-функция не является взаимно однозначной: одному хэшу может соответствовать много объектов. Такие функции называют *сюръективными*.

Для некоторых задач удобнее работать с хэшами, чем с самими объектами. Пусть даны n строк длины m, и нас просят q раз проверять произвольные две на равенство. Вместо наивной проверки за $O(q \cdot n \cdot m)$, мы можем посчитать хэши всех строк, сохранить, и во время ответа на запрос сравнивать два числа, а не две строки.



Применения в реальной жизни

- **Чек-суммы**. Простой и быстрый способ проверить целостность большого передаваемого файла — посчитать хэш-функцию на стороне отправителя и на стороне получателя и сравнить.
- **Хэш-таблица**. Класс `unordered_set` из STL можно реализовать так: заведём n изначально пустых односвязных списков. Возьмем какую-нибудь хэш-функцию f с областью значений $[0, n)$. При обработке `.insert(x)` мы будем добавлять элемент x в $f(x)$-тый список. При ответе на `.find(x)` мы будем проверять, лежит ли x-тый элемент в $f(x)$-том списке. Благодаря «равномерности» хэш-функции, после k добавлений ожидаемое количество сравнений будет равно $\frac{k}{n} = O(1)$ при правильном выборе n.
- **Мемоизация**. В динамическом программировании нам иногда надо работать с состояниями, которые непонятно как кодировать, чтобы «разгладить» в массив. Пример: шахматные позиции. В таком случае нужно писать динамику рекурсивно и хранить подсчитанные значения в хэш-таблице, а для идентификации состояния использовать его хэш.
- **Проверка на изоморфизм**. Если нам нужно проверить, что какие-нибудь сложные структуры (например, деревья) совпадают, то мы можем придумать для них хэш-функцию и сравнивать их хэши аналогично примеру со строками.
- **Криптография**. Правильнее и безопаснее хранить хэши паролей в базе данных вместо самих паролей — хэш-функцию нельзя однозначно восстановить.
- **Поиск в многомерных пространствах**. Детерминированный поиск ближайшей точки среди m точек в n-мерном пространстве быстро не решается. Однако можно придумать хэш-функцию, присваивающую лежащим рядом элементам одинаковые хэши, и делать поиск только среди элементов с тем же хэшом, что у запроса.

Хэшируемые объекты могут быть самыми разными: строки, изображения, графы, шахматные позиции, просто битовые файлы.

Сегодня же мы остановимся на строках.

Полиномиальное хэширование

Лайфхак: пока вы не выучили все детерминированные строковые алгоритмы, научитесь пользоваться хэшами.

Будем считать, что строка — это последовательность чисел от 1 до m (размер алфавита). В C++ `char` это на самом деле тоже число, поэтому можно вычитать из символов минимальный код и кастовать в число: `int x = (int) (c - 'a' + 1)`.

Определим *прямой полиномиальный хэш* строки как значение следующего многочлена:

$$h_f = (s_0 + s_1 k + s_2 k^2 + \ldots + s_n k^n) \mod p$$

Здесь k — произвольное число больше размера алфавита, а p — достаточно большой модуль, вообще говоря, не обязательно простой.

Его можно посчитать за линейное время, поддерживая переменную, равную нужной в данный момент степени k:

```
const int k = 31, mod = 1e9+7;

string s = "abacabadaba";
long long h = 0, m = 1;
for (char c : s) {
    int x = (int) (c - 'a' + 1);
    h = (h + m * x) % mod;
    m = (m * k) % mod;
}
```

Можем ещё определить *обратный полиномиальный хэш*:

$$h_b = (s_0 k^n + s_1 k^{n-1} + \ldots + s_n) \mod p$$

Его преимущество в том, что можно написать на одну строчку кода меньше:

```
long long h = 0;
for (char c : s) {
    int x = (int) (c - 'a' + 1);
    h = (h * k + x) % mod;
}
```

Автору проще думать об обычных многочленах, поэтому он будет везде использовать прямой полиномиальный хэш и обозначать его просто буквой h.

Зачем это нужно?

Используя тот факт, что хэш это значение многочлена, можно быстро пересчитывать хэш от результата выполнения многих строковых операций.

Например, если нужно посчитать хэш от конкатенации строк a и b (т. е. b приписали в конец строки a), то можно просто хэш b домножить на $k^{|a|}$ и сложить с хэшом a:

$$h(ab) = h(a) + k^{|a|} \cdot h(b)$$

Удалить префикс строки можно так:

$$h(b) = \frac{h(ab) - h(a)}{k^{|a|}}$$

А суффикс — ещё проще:

$$h(a) = h(ab) - k^{|a|} \cdot h(b)$$

В задачах нам часто понадобится домножать k в какой-то степени, поэтому имеет смысл предпосчитать все нужные степени и сохранить в массиве:

```
const int maxn = 1e5+5;

int p[maxn];
p[0] = 1;

for (int i = 1; i < maxn; i++)
    p[i] = (p[i-1] * k) % mod;
```

Как это использовать в реальных задачах? Пусть нам надо отвечать на запросы проверки на равенство произвольных подстрок одной большой строки. Подсчитаем значение хэш-функции для каждого префикса:

```
int h[maxn];
h[0] = 0; // h[k] -- хэш префикса длины k

// будем считать, что s это уже последовательность int-ов

for (int i = 0; i < n; i++)
    h[i+1] = (h[i] + p[i] * s[i]) % mod;
```

Теперь с помощью этих префиксных хэшей мы можем определить функцию, которая будет считать хэш на произвольном подотрезке:

$$h(s[l:r]) = \frac{h_r - h_l}{k^l}$$

Деление по модулю возможно делать только при некоторых `k` и `mod` (а именно — при взаимно простых). В любом случае, писать его долго, и мы это делать не хотим.

Для нашей задачи не важно получать именно полиномиальный хэш — главное, чтобы наша функция возвращала одинаковый многочлен от одинаковых подстрок. Вместо приведения к нулевой степени приведём многочлен к какой-нибудь достаточно большой — например, к n-ной. Так проще — нужно будет домножать, а не делить.

$$\hat{h}(s[l:r]) = k^{n-l}(h_r - h_l)$$

```
int hash_substring (int l, int r) {
    return (h[r+1] - h[l]) * p[n-l] % mod;
}
```

Теперь мы можем просто вызывать эту функцию от двух отрезков и сравнивать числовое значение, отвечая на запрос за $O(1)$.

Упражнение. Напишите то же самое, но используя *обратный* полиномиальный хэш — этот способ тоже имеет право на существование, и местами он даже проще. Обратный хэш подстроки принято считать и использовать в стандартном виде из определения, поскольку там нет необходимости в делении.

Лайфхак. Если взять обратный полиномиальный хэш короткой строки на небольшом алфавите с $k = 10$, то числовое значение хэша строки будет наглядно соотноситься с самой строкой:

$$h(abacaba) = 1213121$$

Этим удобно пользоваться при дебаге.

Примеры задач

Количество разных подстрок. Посчитаем хэши от всех подстрок за $O(n^2)$ и добавим их все в `std::set`. Чтобы получить ответ, просто вызовем `set.size()`.

Поиск подстроки в строке. Можно посчитать хэши от шаблона (строки, которую ищем) и пройтись «окном» размера шаблона по тексту, поддерживая хэш текущей подстроки. Если хэш какой-то из этих подстрок совпал с хэшом шаблона, то мы нашли нужную подстроку. Это называется алгоритмом Рабина-Карпа.

Сравнение строк (больше-меньше, а не только равенство). У любых двух строк есть какой-то общий префикс (возможно, пустой). Сделаем бинпоиск по его длине, а дальше сравним два символа, идущие за ним.

Палиндромность подстроки. Можно посчитать два массива — обратные хэши и прямые. Проверка на палиндром будет заключаться в сравнении значений `hash_substring()` на первом массиве и на втором.

Количество палиндромов. Можно перебрать центр палиндрома, а для каждого центра — бинпоиском его размер. Проверять подстроку на палиндромность мы уже умеем. Как и всегда в задачах на палиндромы, случаи четных и нечетных палиндромов нужно обрабатывать отдельно.

Хранение строк в декартовом дереве

Если для вас всё вышеперечисленное тривиально: можно делать много клёвых вещей, если «оборачивать» строки в декартово дерево. В вершине дерева можно хранить символ, а также хэш подстроки, соответствующей её поддереву. Чтобы поддерживать хэш, нужно просто добавить в `upd()` пересчёт хэша от конкатенации трёх строк — левого сына, своего собственного символа и правого сына.

Имея такое дерево, мы можем обрабатывать запросы, связанные с изменением строки: удаление и вставка символа, перемещение и переворот подстрок, а если дерево персистентное — то и копирование подстрок. При запросе хэша подстроки нам, как обычно, нужно просто вырезать нужную подстроку и взять хэш, который будет лежать в вершине-корне.

Если нам не нужно обрабатывать запросы вставки и удаления символов, а, например, только изменения, то можно использовать и дерево отрезков вместо декартова.

Вероятность ошибки и почему это всё вообще работает

У алгоритмов, использующих хэширование, есть один неприятный недостаток: недетерминированность. Если мы сгенерируем бесконечное количество примеров, то когда-нибудь нам не повезет, и программа отработает неправильно. На CodeForces даже иногда случаются взломы решений, использующих хэширование — можно в оффлайне сгенерировать тест против конкретного решения.

Событие, когда два хэша совпали, а не должны, называется *коллизией*. Пусть мы решаем задачу определения количества различных подстрок — мы добавляем в `set` $O(n^2)$ различных случайных значений в промежутке $[0, m)$. Понятно, что если произойдет коллизия, то мы какую-то строку не учтем и получим WA. Насколько большим следует делать m, чтобы не бояться такого?

Выбор констант

Практическое правило: если вам нужно хранить n различных хэшей, то безопасный модуль — это число порядка $10 \cdot n^2$. Обоснование — см. парадокс дней рождений.

Не всегда такой можно выбрать один — если он будет слишком большой, будут происходить переполнения. Вместо этого можно брать два или даже три модуля и считать много хэшей параллельно.

Можно также брать модуль 2^{64}. У него есть несколько преимуществ:

- Он большой — второй модуль точно не понадобится.
- С ним ни о каких переполнениях заботиться не нужно — если все хранить в `unsigned long long`, процессор сам автоматически сделает эти взятия остатков при переполнении.
- С ним хэширование будет быстрее — раз переполнение происходит на уровне процессора, можно не выполнять долгую операцию `%`.

Всё с этим модулем было прекрасно, пока не придумали тест против него. Однако, его добавляют далеко не на все контесты — имейте это в виду.

В выборе же k ограничения не такие серьезные:

- Она должна быть чуть больше размера словаря — иначе можно изменить две соседние буквы и получить коллизию.
- Она должна быть взаимно проста с модулем — иначе в какой-то момент всё может занулиться.

Главное — чтобы значения k и модуля не знал человек, который генерирует тесты.

Парадокс дней рождений

> В группе, состоящей из 23 или более человек, вероятность совпадения дней рождения хотя бы у двух людей превышает 50%.

Более общее утверждение: в мультимножество нужно добавить $\Theta(\sqrt{n})$ случайных чисел от 1 до n, чтобы какие-то два совпали.

Первое доказательство (для любителей матана). Пусть $f(n, d)$ это вероятность того, что в группе из n человек ни у кого не совпали дни рождения. Будем считать, что дни рождения распределены независимо и равномерно в промежутке от 1 до d.

$$f(n, d) = (1 - \frac{1}{d}) \times (1 - \frac{2}{d}) \times \ldots \times (1 - \frac{n-1}{d})$$

Попытаемся оценить f:

$$\begin{aligned}
e^x &= 1 + x + \frac{x^2}{2!} + \ldots && \text{(ряд Тейлора для экспоненты)} \\
&\simeq 1 + x && \text{(аппроксимация для } |x| \ll 1) \\
e^{-\frac{n}{d}} &\simeq 1 - \frac{n}{d} && \text{(подставим } \frac{n}{d} \ll 1) \\
f(n, d) &\simeq e^{-\frac{1}{d}} \times e^{-\frac{2}{d}} \times \ldots \times e^{-\frac{n-1}{d}} \\
&= e^{-\frac{n(n-1)}{2d}} \\
&\simeq e^{-\frac{n^2}{2d}}
\end{aligned}$$

Из последнего выражения более-менее понятно, что вероятность $\frac{1}{2}$ достигается при $n \approx \sqrt{d}$ и в этой точке изменяется очень быстро.

Второе доказательство (для любителей теорвера). Введем $\frac{n(n-1)}{2}$ индикаторов — по одному для каждой пары людей (i, j) — каждый будет равен единице, если дни рождения совпали. Ожидание и вероятность каждого индикатора равна $\frac{1}{d}$.

Обозначим за X число совпавших дней рождений. Его ожидание равно сумме ожиданий этих индикаторов, то есть $\frac{n(n-1)}{2} \cdot \frac{1}{d}$.

Отсюда понятно, что если $d = \Theta(n^2)$, то ожидание равно константе, а если d асимптотически больше или меньше, то X стремится нулю или бесконечности соответственно.

Примечание: формально, из этого явно не следует, что вероятности тоже стремятся к 0 и 1.

Бонус: «мета-задача»

Дана произвольная строка, по которой известным только авторам задачи способом генерируется ответ yes/no. В задаче 100 тестов. У вас есть 20 попыток отослать решение. В качестве фидбэка вам доступны вердикты на каждом тесте. Вердиктов всего два: OK (ответ совпал) и WA. Попытки поделить на ноль, выделить терабайт памяти и подобное тоже считаются как WA.

«Решите» задачу.