Алгоритмы и структуры данных. Домашнее задание №3

Выполнил студент ПМИ Наседкин Дмитрий Сергеевич (группа 242)

Письменная часть

№ 1

Обозначм переменные $a_1, a_2, \ldots, a_n$, тогда сумма $a_1 + a_2 + \cdots + a_n = S_n$.

Давайте докажем, что $S_n \ mod \ p$ равномерно распределена, тогда из этого следует, что $p(S_n \ mod \ p = r) = \frac{1}{p}$, так как у нее всего p значений. И тогда очевидно, что $p(S_n = X) \leq \frac{1}{p}$, так как любая сумма S_n имеет конкретный остаток, а значит $p(S_n = X) \leq p(S_n \ mod \ p = X \ mod \ p) = \frac{1}{p}$.

Док-во: Докажем по индукции по n:

База $n = 1$: Следует из условия на каждую переменную, очев.

Переход: Пусть при n переменных $p(S_n \ mod \ p = r) = \frac{1}{p}$ (из равномерности), тогда:

$$p(S_{n+1} \, mod \, p = r) = \sum_{i=0}^{p-1} p(S_n \, mod \, p = (r-i) \, mod \, p) * p(a_{n+1} = i) = \sum_{i=0}^{p-1} \frac{1}{p^2} = \frac{p}{p^2} = \frac{1}{p},$$

то есть равномерно распределена.

№ 3

Заранее оговорю, что и в пунке а и в б считаем всегда по модулю $10^9 + 7$, а p - это какое нибудь нечетное число, большее размера алфавита.

Пункт а:

Идея:

Если у строки i, j совпадают с точностью до циклического сдвига, то можно зафиксировать строку i и рассмотреть все циклические сдвиги j, тогда хотя бы раз строка i будет равна какому-то циклическому сдвигу j, то строки равны с точностью до циклического сдвига, и не равны иначе.

Предподсчет за $O(L)$:

Давайте для каждой строки насчитаем хеши всех ее циклических сдвигов и сохраним в хеш-таблицу(`unordered_set`). Для этого насчитаем хеш нашей строки, он будет равен $s_1 * p^{|s|-1} + s_2 * p^{|s|-2} + \cdots + s_{|s|}$ и после постепенно будем вычислять хеш каждого циклического сдвига. Пусть вычислен хеш $hash$ циклического сдвига, если начать с i-ой позиции, тогда хеш циклического сдвига с $i+1$-ой позиции будет равен $(hash - s_i * p^{|s|-1}) * p + s_i$. Все степени p можно заранее предпосчитать поэтому переход делается за $O(1)$. Тогда для строки длины l хеши всех циклических сдвигов считаются за $O(l)$, а значит весь предпосчет работает за $O(L)$ по времени и так как храним все циклические сдвиги для каждой строки в виде хешей, то затраченная память тоже $O(L)$.

Ответы на запросы за $O(1)$:

Для ответа на запрос возьмем любой(например первый из хеш-таблицы для i-ой строки) хеш строки i, и проверим существует ли такой же хеш в хеш-таблице для j, так как там были все циклические сдвиги j, то если найдется, то ответ - "да иначе - "нет". Запрос в хеш-таблицу делается за $O(1)$, поэтому время ответа на запрос будет такое же.

Устная и письменная часть

№ 3

Пункт б:

Улучшим нашу идею, заметим, что, если строки i, j совпадают с точностью до циклического сдвига, то и минимальные хешы каждой строки тоже равны, и обратное также верно. То есть просто будем поддерживать минимальный хеш среди всех циклических сдвигов строки, а не запоминать их все(при этом предпосчет p точно также понадобиться и максимальная нужная степень p равна $max|s_i|$), то есть для каждой строки будем хранить только минимальный хеш. Итого по памяти станет $O(n + max|s_i|)$. На запрос же будем просто сравнивать вычисленные минимальные хеши для i и j.

№ 4

Идея:

Давайте дополнительно хранить вектор пар, который поможет нам отличать ячейки в большом массиве, где написаны наши данные, а где "мусор". Парой будет является - {ключ, значение}.

Большой массив назовем a, вспомогательный вектор q.

Запросы:

- Инциализация - просто обьявим наш вспомогательный вектор пар q.
- Запрос вставки - Пусть запрос имеет номер i(нумерация с 0, при этом отсчитываем только запросы добавления), тогда присвоим a_{key} значение i. В конец вектора q добавим пару {ключ, значение}, то есть $q_i = \{key, value\}$.
- Запрос поиска - Для этого просто посмотрим, что хранится в a_{key}, тогда если в $q_{a_{key}}.first == key$, то такой элемент есть, можно дальше спокойно вывести $value$. Потому что если это мусорное значение, то мы или выйдем за границы вектора, или первый элемент пары укажет не ту ячейку.
- Запрос удаления - В этом случае присвоим a_{key} значение -1. То есть фактически оно станет мусорным, так как при очередном запросе поиска он выйдет за границу q, то есть элемент действительно удален.

Очевидно, что все операции работают за $O(1)$, а памяти требуется столько же, сколько будет запросов добавления, то есть $O(n)$, если запросов n. Каждый хранимый обьект занимает одну ячейку в a и одну в q, то есть $O(1)$ памяти на каждый обьект.

№ 5

Давайте создадим хеш от ациклического графа (`h[v]`), тогда `h[v] == h[u]` тогда и только тогда, когда `L[u] == L[v]`. Научимся строить хеш с таким условием(вероятности ошибки нет).

Будем считать `h[v]` в обрятном порядке топ-сорта(чтобы когда считаем `h[v]` то для всех достижимых уже было посчитано, топсорт работает за $O(n + m) = O(m)$). Пусть `h[v] = 0` для любой листовой вершины, так как `L[v] = L[u]` для любых двух листовых вершин. Тогда посмотрим на вектор пар {буква, написанная на ребре, `h[u]` вершины, в которую ведет ребро}, тогда если `L[u] == L[v]`, то соответствующие векторы пар также равны с точностью до сортировки. (Это очевидно и легко доказать по индукции, пусть для всех предыдущих верно, тогда для двух вершин это также верно, если мы переходим по одинаковым буквам в одинаковые множества).

Заведем хеш мапу, в которой ключ — это вектор пар переходов, а значение — `h` для всех вершин у которых переходы именно такие. Поскольку суммарный размер всех векторов пар по всем вершинам равен $O(m)$, и поиск и вставка в хеш-мапе может быть сделана за $O( )$, время работы будет $O(m)$(так как размер алфавита фиксированный, то можно считать что сортировка работает за константу).

Как узнать хеш-вершины если посчитан вектор пар переходов для нее? Посмотрим на вектор пар переходов:

- Вершина с таким ключом есть, тогда присваиваем этой вершине тот же `h`.
- Вершин с таким ключом нет, тогда заносим в хеш-мапу данную пару (вектор пар и вершина) и `h[v] := sz от размера хеш-мапы`.

В конце остается проверить, что `h[s] == h[t]`.

№ 6

Подготовка:

Для начала научимся сравнивать 2 мультимножества без дополнительной памяти. Воспользуемся идеей хеширования мультимножеств через сумму, когда для каждого элемента мы запомнили какое-то случайное значение, а после суммировали. Тогда если хеши мультимножеств совпали, то значит и мультимножества равны. Попробуем обойтись без лишней памяти.

Как мы знаем, для генератора псевдослучайных чисел нужен *seed*, после чего он переиспользует старые значения как новый сид для следующего. Но и ничего не мешает после каждой операции менять сид, и тогда можно просто взять в качестве *seed* само значение числа, от которого мы хотим получить случайное, тогда сколько бы раз мы не вызвали от какого-то числа получение случайного значения, каждый раз оно будет одно и то же, а это то, что мы и хотели сделать. (Для препятствия взлому можно, например, добавлять к любому значению фиксированное k, которое вначале выбрать от времени начала программы). Очевидно, что теперь мы используем $O(1)$ доп. памяти, а время работы осталось тем же, то есть $O(n)$.

Алгоритм:

Для начала сравним наш массив a и "идеальный массив"$\{1, 2, 3, ..., n\}$, если они равны как мультимножества, то все элементы различны, иначе есть дупликаты. Для удобства обьяснения введем массив распределения чисел, то есть cnt_i - кол-во чисел со значением i. Пусть мы разделили какой-то подотрезок cnt на 2 части, где есть дупликат, тогда хотя бы одна из них не совпадет как мультимножества с "идеальным"(иначе дупликатов нет), для этого из a будем рассматривать только элементы в диапозоне от l до r, а "идеальное"посчитать можно явно то есть массив $\{l, l+1, l+2, \ldots, r\}$, итого за $O(n)$. Если несовпадение только в одной части, то в ней же очевидно дупликат. То есть осталось отличать случаи, когда в обоих частях не совпадают с идеальным, для этого посчитаем сколько элементов в каждой из частей(то есть сумма по массиву cnt на отрезке), и очевидно, что дупликат точно находится в той части, где сумма больше(или в обоих, если равны), действительно, в большей по сумме части есть хотя бы одна 2, так как иначе она бы не была большей. Значит всегда можно определить в какой из частей дупликат, а значит можно запустить бинпоиск. При этом будет $\log n$ итераций определения какая из частей содержит дупликат, а значит алгоритм работает за $O(n \log n)$.

№ 7

Для начала случайно пошафлим точки, выберем первые 2 точки, назовем их x_1, x_2, тогда если они равны, то можно завершаться(ответ 0), иначе назовем d_k - минимальное кратчайшее растояние на если рассмотреть первые k точек, то есть $d_2 = x_2 - x_1$. Тогда для перехода на $k+1$ шаг разделим нашу прямую на блоки длиной d_k, номером блока будет $\frac{x_i}{d_k}$. При этом в каждом блоке будет не более 1-ого числа(иначе d_k не минимальное расстояние), для этого удобно хранить блоки как хеш-таблицу. Добавим $k+1$ точку, тогда возможны 2 варианта:

- Она обновила текущий минимум(чтобы это понять, проверим что точка попала в блок где уже была другая точка, либо посмотрим на соседние 2 блока, так как кратчайшее расстояние могло обновиться и между соседними блоками, делается за $O(1)$ запросом в хеш-таблицу), в этом случае перестроим хеш-таблицу за $O(k)$. (Если $d_{k+1} = 0$, то можно завершиться)
- И не обновила иначе, тогда перестраивать структуру не надо.

Посчитаем ожидаемое время работы алгоритма, оно равно $\sum_{k=3}^{n} p(k) *$

$O(k)$, где $p(k)$ - вероятность того, что на k-ом шаге придется перестраивать структуру, то есть что наша точка стала одной из двух, на которой достигается минимальное расстояние, если $p(k) = \frac{2(k-1)}{k*(k-1)} = \frac{2}{k}$(если минимальных расстояний несколько, то или обновлять структуру не надо, а если с нашей точкой, то вероятность еще меньше - $\frac{1}{k}$). Тогда:

$$\sum_{k=3}^{n} p(k) * O(k) = \sum_{k=3}^{n} \frac{2}{k} * O(k) = \sum_{k=3}^{n} O(1) = O(n)$$