Алгоритмы
Домашнее задание №3
Стрелец Владимир

Задача 2.

Заметим, что если строка имеет вид "111...1100...000", то вне зависимости от количества нулей и количества единиц её суф. массив будет иметь вид

n, n-1, n-2, … , 1 и тогда однозначно восстановить строку не получится.

Пусть строка имеет другой вид. Рассмотрим любую последовательность нулей вместе со следующей единицей (если она имеется). Числа-суффиксы этой последовательности в суффиксном массиве расположены в таком же порядке (только между ними могут быть другие числа).
Рассмотрим любую последовательность единиц исходной строки вместе со следующим нулём (если он есть). Суффиксы этой последовательности расположены в суф. массиве в обратном порядке.

Таким образом, если suf[i] = k, причём (k - 1) встречался раньше в суф. массиве, то в исходной строке на месте s[k - 1] должен стоять 0, иначе -- 1.

Воспользуемся этими наблюдениями. Заведём массив флагов was размером таким же, как и суффиксный массив. В нём на месте was[j] будем хранить, было ли встречено число j ранее при последовательном просмотре суф. массива.

Итак, можно сначала проверить, не имеет ли суф. массив вид n, n-1, … , 1. Если именно так, то строку восстановить не получится
Иначе пробегаемся ещё раз по суф. массиву.
Если suf[i] = k, то ставим was[k] = true.
Затем, если $k \neq 1$, если was[k - 1] == true, то ставим s[k - 1] = 0,
если was[k - 1] == false, то ставим s[k - 1] = 1.

Таким образом, останется определиться с последним символом. Если он был нулём, то в суф. массиве он будет первым (так как это минимальная возможная подстрока в бинарной строке). Итак, первой проверкой (равны ли n и suf[1]) мы определимся и с последним символом.

Задача 4.

Для каждого элемента a[k] массива введём две величины:
1) длина наибольшей возрастающей подпоследовательности, заканчивающейся в a[k] -- val_1;
2) длина наибольшей убывающей подпоследовательности, начинающейся в a[k] -- val_2.

Теперь рассмотрим два любых элемента массива a[k] и a[l].
Если a[k] > a[l], то a[k].val_2 > a[l].val_2 (наибольшая убывающая подпоследовательность, начинающаяся в a[l] будет входить в наибольшую убывающую подпоследовательность, начинающейся в a[k]. А также имеется сам элемент a[k]).
Если же a[k] < a[l], то a[k].val_1 < a[l].val_1 (по аналогичным соображениям).

Таким образом, получим
УТВЕРЖДЕНИЕ: "для любых двух элементов последовательности не будет обе введённые величины не могут быть равными".
И тогда если предположить, что все возрастающие подпоследовательности имеют длину не большую n, а все убывающие -- длину не больше m, то всего различных классов эквивалентностей по паре (val_1, val_2) в массиве может быть лишь mn. И тогда по принципу Дирихле получим, что в одном из классов будут два элемента. Что противоречит утверждению. Значит, в массиве есть либо убывающая подпоследовательность длины m + 1, либо возрастающая длины n + 1.

Задача 7.

https://contest.yandex.ru/contest/1080/run-report/604680/

Задача 8.

https://contest.yandex.ru/contest/1080/run-report/605075/
Нужно найти для каждого i максимальное k, такое что есть j < i, что
s[i + k - 1] = s[j + k - 1].

Алгоритм:
для этого будем рассматривать последовательно суффиксы Suf_i, i = 1, 2, … , n и для каждого из них находить такое k.
Построим суф-массив, lcp-массив, а на основе lcp-массива -- дерево отрезков для минимума. И теперь нам нужно для каждого суффикса найти один из

прошлых суффиксов (в исходной строке), такой что их lcp будет наибольшим из всех возможных. Так как $lcp(i,j) = \min[i \le k < j]\ lcp(k,k+1)$, то нужно находить наиближайшие в суф-массиве суффиксы слева и справа в суф-массиве (но прошлые по исходной строке) и сравнивать их lcp. Для этого воспользуемся set'ом и его функцией upper_bound, которая найдёт за log |S| первый больший элемент среди добавленных. А добавлять в set будем позиции суффиксов при их последовательном просмотре в исходной строке.
Для этого введём массив positions, такой что для суф-массива suf выполнено: positions[suf[j]] = j для всех j = 1, 2, … , n.

Итак, сложность алгоритма:

1) построение суф-массива -- O(|S| log |S|)
2) построение lcp-массива -- O(|S|)
3) построение массива positions -- O(|S|)
4) построение дерева отрезков для lcp-массива -- O(|S|)
5) А далее на каждой из |S| итераций будем за O(log |S|) получать ближайшие слева и справа по суф-массиву прошлые по исходной строке суффиксы для Suf_i, i = 1, 2, … , n с помощью функции upper_bound у set'a (в set'е -- позиции всех прошлых суффиксов в суф-массиве).

Для этого добавим в set число, заведомо большее всех остальных (например, |S| + 1) и будем иметь ввиду, что если upper_bound вернула итератор на |S| + 1, то это значит, что суффиксов с большими позициями ранее не встретилось. Также поможет информация о том, что --upperbound -- ближайший суффикс с меньшей позицией.
Итак, найдя позиции интересуемых суффиксов, запросим у дерева отрезков минимум для нахождения наибольшего общего префикса интересующих строк. Итого, |S| итераций по O(log |S|) времени на каждую.

Итоговая сложность -- O(|S| log |S|).

Что-то пошло не так и код не заработал. Но на данных тестах выдаёт правильный ответ.