Разбор задачи «ННД»

Подзадачи 1 и 2

В подзадачах 1 и 2 найти наибольший нечётный делитель (ННД) нужно только у одного числа n ($n = l_1 = r_1$). Для этого достаточно написать перебор делителей числа n с асимптотикой $\mathcal{O}(n)$ или $\mathcal{O}(\sqrt{n})$ соответственно.

Подзадачи 3 и 4

Заметим, что ННД числа n можно выразить как:

$$\text{ННД}(n) = \frac{n}{2^k}$$

где 2^k — максимальная степень двойки, на которую нацело делится число n.

Получается, что вычислить ННД(n) можно жадным алгоритмом («делить n на 2 пока делится») с асимптотикой $\mathcal{O}(\log n)$ или с помощью битовых операций (ННД$(n) = \dfrac{n}{n - (n \,\&\, (n-1))}$ или ННД$(n) = n >>$ __builtin_ctzll(n)) с асимптотикой $\mathcal{O}(1)$ соответственно.

Для нахождения ответа в подзадачах 3 и 4 остаётся суммировать ННД(n) по всем n от l_1 до r_1.

Подзадача 5

Для решения подзадачи 5 можно воспользоваться идеей динамического программирования и предподсчитать массив префиксных сумм $pref$ такой, что $pref[i] = \text{ННД}(1) + \text{ННД}(2) + \ldots + \text{ННД}(i) = pref[i-1] + \text{ННД}(i)$. После чего ответ на i-й вопрос можно вычислить как $pref[r_i] - pref[l_i - 1]$.

Асимптотика такого решения — $\mathcal{O}(\max(r_i) + t)$.

Подзадача 6 и полное решение

Вместо того чтобы вычислять ННД для каждого числа отдельно, можно перебирать степени двойки и для каждой степени считать, сколько чисел имеют именно эту степень в выражении для расчёта ННД.

Таким образом, для каждого k достаточно найти количество чисел, принадлежащих диапазону от l_i до r_i, вида $m \cdot 2^k$, где m — нечётное число. Наименьшее такое число, если такое существует, равно $m_{\min} \cdot 2^k$, где $m_{\min} = \left\lceil \dfrac{l}{2^k} \right\rceil$, если $\left\lceil \dfrac{l}{2^k} \right\rceil$ — нечётное число, а иначе — $m_{\min} = \left\lceil \dfrac{l}{2^k} \right\rceil + 1$. Наибольшее такое число, если такое существует, равно $m_{\max} \cdot 2^k$, где $m_{\max} = \left\lfloor \dfrac{r}{2^k} \right\rfloor$, если $\left\lfloor \dfrac{r}{2^k} \right\rfloor$ — нечётное число, а иначе — $m_{\max} = \left\lfloor \dfrac{r}{2^k} \right\rfloor - 1$.

Сумму всех ННД для данного k можно посчитать по формуле суммы арифметической прогрессии:

$$\frac{m_{\min} + m_{\max}}{2} \cdot \left(\frac{m_{\max} - m_{\min}}{2} + 1 \right)$$

Асимптотика такого решения — $\mathcal{O}(t \cdot \log(\max(r_i)))$.

Решения, в которых случается переполнение 64-битного типа данных, например, из-за взятия остатка при делении на $10^9 + 9$ только перед выводом ответа на вопрос, проходят тесты подзадачи 6.

Разбор задачи «Тотальная неоднозначность»

Заметим, что если мы можем разбить массив на префикс длины k и суффикс длины $n-k$, так что оба они являются перестановками, мы однозначно можем заключить, что максимальный элемент

массива a равен $\max(k, n-k)$. Значит, если есть хотя бы два разбиения на префикс суффикс размеров $(k, n-k)$ и $(l, n-l)$, то должно быть выполнено $\max(k, n-k) = \max(l, n-l)$, так как глобальная характеристика вида максимум массива никак не меняется. И значит, единственный вариант когда это может быть возможно, если $k+l = n$, то есть длины отрезков должны совпадать при разбиении. Поэтому множество неоднозначных массивов длины n получается очень ограниченным и имеет следующее описание:

«Перестановка чисел от 1 до k» + «Перестановка чисел от $k+1$ до $n-k$» + «Перестановка чисел от $n-k+1$ до n», для некоторого $1 \leqslant k \leqslant \frac{n-1}{2}$, здесь k как раз размер префикса в одном из разбиений. И всего есть два разбиения: $(k, n-k)$ и $(n-k, k)$.

Далее, хотим для всех k понять минимальное число операций, чтобы массив принял такой вид. После чего минимум из этих величин будет ответом на задачу.

Мы следим за 3 отрезками: префикс длины k, суффикс длины k, и серединка длины $n-2k$.

Фактически мы хотим на префиксе и суффиксе поддерживать количество различных встречающихся чисел от 1 до k, а для серединке количество различных встречающихся чисел от $k+1$ до $n-2k$.

Явно посчитаем эти величины для $k = 1$. А далее будем итеративно увеличивать k. С каждым увеличением k на один; каждый из трёх отрезков немного сдвигает свои границы на ± 1 влево и вправо, и отрезок значений за которыми мы следим также сдвигается на ± 1. Для каждого из 3 отрезков храним свой массив подсчёта, помечающий какие элементы сейчас принадлежат отрезку. Также несложно и поддерживать количество различных встречающихся чисел на интересном отрезке значений, в отдельной переменной. При аккуратной реализации всё это работает за $O(n)$.

Разбор задачи «Слон Филимон и Очень Важное Сообщение»

Задачу можно переформулировать как: «Сколько есть способов разбить массив на подотрезки так, чтобы каждый подотрезок можно было сжать в один элемент используя операции», так как каждый способ разбиения на подотрезки задаёт один из возможных конечных массивов, и соответственно каждый конечный массив задаётся разбиением на подотрезки.

Будем решать задачу постепенно улучшая асимптотику.

Решение за $O(n^3)$

Мы хотим для каждого подотрезка $a_l, a_{l+1}, \ldots, a_r$ понять — а можно ли сжать этот отрезок в один элемент? Будем делать это техникой динамического программирования по подотрезкам, пусть $can[l][r]$ — булевое значение: можно ли сжать $a_l, \ldots, a_r$ в один элемент.

База динамики: $can[l][l] = true$ для всех l, действительно один элемент сжимается в один элемент.

Переход: хотим узнать $can[l][r]$. Пусть отрезок сжимается в один элемент, тогда давайте переберём какой могла быть последняя операция. Мы точно знаем, что она имела вид объединения двух чисел $a_l + \ldots + a_{mid}$ и $a_{mid+1}, \ldots, a_r$ для некоторого $l \leqslant mid < r$. Поэтому можем перебрать mid в этих границах и проверить: можно ли сжать $a_l, \ldots, a_{mid}$ в один элемент, просто посмотрев на уже насчитанное значение $can[l][mid]$, и аналогично с $a_{mid+1}, \ldots, a_r$ посмотрев на значение $can[mid+1][r]$. И также надо проверить, является ли простым число $a_l + \ldots + a_r$: чтобы последняя операция была валидной. Мы можем совершать проверку на простоту за $O(1)$, заранее насчитав решето Эратосфена размера $33 \cdot n$.

Так мы можем насчитать массив $can[l][r]$ за $O(n^3)$.

После чего посчитать количество достижимых массивов можно второй динамикой по префиксу. Пусть $dp[i]$ — ответ для $a_1, a_2, \ldots, a_i$.

База: $dp[0] = 1$ (пустой массив)

Переход: $dp[i] = \sum_{j=0}^{i-1} dp[j] \cdot can[j+1][i]$, буквально перебираем все возможные варианты последнего отрезка, и для тех у кого $can[j+1][i] = true$ добавляем $dp[i][j]$.

Эта динамика работает за $O(n^2)$ и ответ на задачу находится в $dp[n]$.

Решение за $O(n^2)$

Самое долгое место прошлого решения: необходимость перебирать mid среди $O(n)$ вариантов для каждого пересчёта в первой динамике. Давайте оптимизируем это заметив следующий факт: невозможно такое, что в ходе операций, некоторые два отрезка длины $\geqslant 3$, которые сами по себе сжимаются в один элемент, можно объединить в один. Почему это так: если отрезок имеет длину $\geqslant 3$ и объединяется в один элемент, значит его сумма обязательно есть простое число $\geqslant 3$, а значит и **нечётное** число. Получаем, что если два таких отрезка могут объединится в один, то два нечётных простых числа в сумме дают простое число. А такого не бывает, так как нечётное + нечётное = чётное, а единственное чётное простое число это 2. Поэтому, нет смысла перебирать значения mid такие, что и $a_l, \ldots, a_{mid}$ и $a_{mid+1}, \ldots, a_r$ имеют длину $\geqslant 3$ — такой пересчёт точно будет безуспешен. Значит можно перебрать всего $O(1)$ значений mid: $\{l, l+1, r-2, r-1\}$. Тогда динамика по подотрезкам будет работать за $O(n^2)$, как и всё решение.

Решение за $O(1327 \cdot n)$

Давайте оптимизировать дальше. Если рассмотреть подгруппу с $a_i = 1$, то для её решения надо заметить, что в итоговом массиве могут быть только числа $1, 2, 3, 5, 7$. Логично предположить, что и для $a_i \leqslant 33$, элементы итогового массива не могут быть *слишком большими.*

И действительно: максимальный элемент массива ограничен числом 1327*.*

Это верно, так как числа 1327 *и* $1327+34$ *являются простыми, а между ними простых чисел нет. И так как* $a_i \leqslant 33$*, несложно видеть, что число* > 1327 *в ходе операций получить нельзя: нельзя сложить два простых числа* > 33 *и получить* $\geqslant 1327+34$*, а также нельзя сложить простое с числом* $\leqslant 33$*, чтобы получить* $\geqslant 1327+34$*, таким образом числа* $\geqslant 1327+34$ *недостижимы.*

А значит, мы точно можем сказать, что в динамике по подотрезкам, если $r-l+1 > 1327$*, то* $a_l+\ldots+a_r > 1327$*, а значит* $can[l][r] = false$ *обязательно! И считать динамику для отрезков длин* > 1327 *бессмысленно. Аналогично во второй динамике можем суммировать только последние* 1327 *значений. Итого получаем решение за* $O(1327 \cdot n)$*.*

Решение за $O(264 \cdot n)$

На самом деле константу 1327 *можно улучшить: ведь это ограничение на максимально возможную сумму отрезка, который сжимается в один элемент, а не на максимально возможную длину такого отрезка, а для оптимизации динамики, в качестве константы нам подойдет именно максимально возможная длина.*

И оказывается максимально возможная длина массива сжимаемого в один элемент это 264*. Так как разности между простыми* > 2 *всегда чётны: а значит составные, либо равны* 2*, чтобы получить одно простое число из другого прибавлением: надо либо прибавить чётное число* > 2*, которое не могло никак иначе появиться в массиве, кроме как быть в нём сначала, либо прибавить* 2*, которая могла изначально иметь вид в массиве* $[1, 1]$*, но не более того. А если в какой-то момент в массиве появятся два простых числа* > 2*, он уже никак не сожмётся в один элемент. Поэтому каждая операция либо заменяет* $[1, 1]$ *на* 2*, либо увеличивает одно и то же простое число, сохраняя его простым. Поэтому максимальная длина массива, сжимаемого в один элемент, ограничена {количество простых чисел* $\leqslant 1327$*} + {количество соседних простых чисел* $\leqslant 1327$ *с разницей 2} + 1, что как раз и есть* 264*. Такое решение уже заходит на полный балл.*

Разбор задачи «Палиндром Цезаря»

Примечание:

Для поиска палиндромов на любом шаге решения можно использовать идею с перестройкой строки. Чтобы не рассматривать случай с чётными и нечётными палиндромами в изначальную строку добавим символы-разделители. Например для строки s = abacaba получим строку t = #a#b#a#c#a#b#a#. Тогда центр палиндрома abba будет находиться в символе # в изменённой строке #a#b#b#a#. Центр палиндрома abcba будет находиться в символе c в изменённой строке #a#b#c#b#a#. Т.е. палиндромы вне зависимости от чётности в изменённой строке становятся палиндромами нечётной длины.

Подгруппа 1.

Напишем переборное решение.

Будем отвечать на запросы независимо. Для каждого запроса переберём всевозможные отрезки изменения и значение сдвига символов.

Для каждого варианта изменения строки проверим какой получим ответ. Для этого в полученной строке переберём центр палиндрома-результата и для каждого центра насчитаем длину наибольшего палиндрома с центром в этой точке.

Получим код:

```
while q--                               # ql, qr - границы очередного запроса
    for l = ql; l <= qr; l++            # l - левая граница отрезка изменения
        for r = l; r <= qr; r++         # r - правая граница отрезка изменения
            for x = 1; x <= k - 1; x++  # x - значение сдвига
                t = shift(s, l, r, x)   # t - изменённая строка
                for c = 0; c < n; c++   # c - центр палиндрома
                    l = 0
                    while t[c-l]==t[c+l]# l - длина палиндрома
                        l++
```

Это решение работает за $O(K * Q * N^4) = O(Q * N * N * K * N * N)$.

Подгруппа 2.

Заменим код поиска палиндромов в строке на алгоритм Манакера (найдёт ответ в изменённой строке за $O(N)$).

Код решения:

```
while q--                               # ql, qr - границы очередного запроса
    for l = ql; l <= qr; l++            # l - левая граница отрезка изменения
        for r = l; r <= qr; r++         # r - правая граница отрезка изменения
            for x = 1; x <= k - 1; x++  # x - значение сдвига
                t = shift(s, l, r, x)   # t - изменённая строка (O(N) действий)
                ans = manaker(t)        # алгоритм Манакера (O(N) действий)
```

Это решение работает за $O(K * Q * N^3) = O(Q * N * N * K * N)$.

Подгруппа 3.

Давайте сначала насчитаем ответы для всевозможных изменений тем же способом.

Тогда ответ на запрос - лучший из достижимых. То есть максимум по всем подотрезкам. Такое можно пересчитать через квадратное динамическое программирование (Пересчитывая от меньших отрезков к большим).

Код решения:

```
# Насчитываем ответ для каждого изменения
for l = ql; l <= qr; l++            # l - левая граница отрезка изменения
    for r = l; r <= qr; r++         # r - правая граница отрезка изменения
        for x = 1; x <= k - 1; x++  # x - значение сдвига
            t = shift(s, l, r, x)   # t - изменённая строка (O(N) действий)
            dp[l][r] = manaker(t)   # алгоритм Манакера (O(N) действий)

# Обновляем dp через подотрезки
for len = 2; len < n; len++
    for l = 0; l < n; l++
        r = l + len - 1
        dp[l][r] = max(dp[l][r], dp[l][r - 1], dp[l + 1][r])
```

```
# Отвечаем на запросы
for [ql, qr] in queries
    ans = dp[ql][qr]
```

Это решение работает за $O(K * N^3 + N^2 + Q) = O(K * N^3)$.

Подгруппа 4.

Давайте поймём как выглядит палиндром в ответе.

Во-первых если сдвиг в ответе пересекал центр палиндрома ответа - элементы слева и справа от центра менялись одинаково (и сдвиг можно было не выполнять в этой части).

Во-вторых если сдвиг происходил справа от центра палиндрома - можно было выполнить противоположный сдвиг в симметричной левой части палиндрома и ответ получился бы тот же.

С этого момента будем рассматривать только сдвиг в правой части палиндрома (левую рассмотрим аналогично в развёрнутой строке).

Для зафиксированного центра палиндрома (в примере это символ 'c') ответ выглядит так:

```
          [abcba] - палиндром, который есть в строке s.
       [###]   [@@@] - здесь выполняем сдвиг (эти символы совпадут при некотором одинаковом сдви
    [zyx]         [xyz] - символы, которые вне сдвига совпадают и без сдвига.
s = ...zyx###abcba@@@xyz...
```

Введём обозначения:

```
Длина [abcba] - len_0. (Изначальный палиндром).
Длина [###] = длина [@@@] - len_1. (Левый и правый shift).
Длина [zyx] = длина [xyz] - len_2.
```

$len_0 + 2 * (len_1 + len_2)$ - наибольшая длина палиндрома, которую возможно получить каким-либо изменением.

Рассмотрим правый shift. Обозначим его границы как l_s и r_s.

Пусть для запроса $[ql, qr]$ выполняется: $(ql \leqslant [l_s, r_s] \leqslant qr)$, то есть shift целиком попадает в запрос. Тогда мы можем полностью выполнить правый shift! И получить ответ: $len_0 + 2 * (len_1 + len_2)$.

Пусть нет. Пусть левая граница shift попадает в запрос. То есть: $(ql \leqslant [l_s \leqslant qr \leqslant r_s])$.

Тогда мы можем выполнить сдвиг на префиксе shift-а. В этом случае только (qr - l_s + 1) символ совпадёт с точки зрения палиндрома.

Тогда мы сможем получить ответ: $len_0 + 2 * (qr - l_s + 1)$.

Пусть нет, то есть левая граница shift не попадает в запрос. Тогда мы никак не сможем привести к равенству даже первый несовпавший символ палиндрома. Значит в этом случае ответ: len_0.

Теперь нужно:

1. Научиться находить len_0, len_1, len_2 для всевозможных центров палиндромов.
2. Учитывать все найденные данные в ответе.

Как найти len_0, len_1, len_2? Переберём центр палиндрома. Найдём перебором сколько символов уже совпадает слева и справа от этого центра. Теперь найдём сколько символов совпадёт дальше при использовании сдвига (значение сдвига однозначно определяется по первому несовпавшему символу). Когда сдвиг перестал помогать символам совпадать - возможно они совпадают и без сдвига. Будем перебирать символы дальше, пока они сопадают.

Пример: Строка s = '..gzyxjlabcbakixyzh...' Рассмотрим центр 'c'. Найдём 'abcba' - палиндром в изначальной строке. Далее справа 'k', слева 'l' $\Rightarrow$ нужно сдвинуть правую часть на 1. Далее справа 'i', слева 'j' $\Rightarrow$ всё ещё поможет тот же сдвиг на 1 в правой части. Далее справа 'xyz', слева 'xyz' $\Rightarrow$ сдвиг на 1 уже не действует, но символы совпадают в s.

```
for c = 0; c < n; c++
    len_0 = len_1 = len_2 = 0
    while совпадение
        len_0++
```

```
x = s[c + len_0] - s[c - len_0]
while сдвиг на x приводит к совпадению
    len_1++
while совпадение
    len_2++
```

Теперь нужно учесть полученные shift-ы в ответе на запросы. Давайте просто переберём для каждого запроса всевозможные центры палиндрома-ответа и пересчитаем возможный ответ через len_0, len_1, len_2 для этого центра.

```
for [ql, qr] in queries
    for [c, len_0, len_1, len_2] in palindrome_info
        if shift внутри [ql, qr]
            ans = len_0 + 2 * (len_1 + len_2)
        else if левая граница shift внутри [ql, qr]
            ans = 2 * (qr - c)
        else
            ans = len_0
```

Получаем решение за $O(N^2 + QN)$

Подгруппа 5.

Работает подход как в подгруппе 4, но теперь число различных запросов не превышает N. Тогда можно для каждой возможной точки изменения сохранить наилучший ответ при изменении в этой точке.

Получим решение за $O(N^2 + Q)$

Подгруппа 6.

Возьмём рассуждения из решения подгруппы 4 и ускорим их до полного решения.

Чтобы узнавать сколько символов подряд совпадает влево от символа l и вправо от символа r достаточно воспользоваться хешами и бинпоиском (например, на строке и её перевёрнутой версии) или построить суффиксный массив на строке и её перевёрнутой версии.

Тот же подход работает и с поиском числа совпадений с учётом сдвига. Тут можно либо искать совпадения в k версиях строки, либо заметить следующее: Совпадение с точностью до сдвига эквивалентно совпадению набора разностей соседних.

Т.е. если:

$$a_0 = b_0 + x$$
$$a_1 = b_1 + x$$
$$a_2 = b_2 + x$$
$$\dots$$

Это то же, что и:

$$a_1 - a_0 = b_1 - b_0$$
$$a_2 - a_1 = b_2 - b_1$$
$$a_3 - a_2 = b_3 - b_2$$
$$\dots$$

Научились находить число совпадающих символов (за $O(\log N)$ для одного центра и за $O(N \log N)$ для всех центров суммарно).

Теперь научимся учитывать полученные отрезки в ответе.

Вспомним, что ответ на запрос зависит от того, попадает ли левая граница очередного shift-а внутрь отрезка запроса.

А так же от позиции правой границы запроса (причём от позиции правой границы запроса зависимость линейная).

Тогда ответ на запрос для shift-а $[l_s, r_s]$ будем хранить в координате l_s.

При передвижении правой границы запроса qr вправо на один:

1. Если qr $< r_s \Rightarrow$ ответ на запрос для этого shift увеличится на 2.
2. Если qr $= r_s \Rightarrow$ ответ на запрос для этого shift станет $(len_0 + 2 * (len_1 + len_2))$ и перестанет увеличиваться.

Идея: Заведём дерево отрезков, в каждой вершине изначально запишем -infinity.

Переберём все значения qr (слева направо). При передвижении qr на один вправо:

1. Ответ во всех точках на отрезке [0, qr] увеличим на 2 (операция прибавления на отрезке).
2. Если был shift с левой границей = новое значение qr $\Rightarrow$ начнём его учитывать.
3. Если был shift с правой границей = старое значение qr $\Rightarrow$ перестанем его учитывать в этом ДО (нам уже нельзя будет увеличивать это значение), положим его в ответ в точке l_s.
4. Если есть запросы с правой границей в текущей qr - ответ на них это max на отрезке [ql, qr].

Осталось всего лишь повторить все действия с деревом отрезков в развёрнутой строке (легко реализуется с помощью использования функций).

Получаем решение за $O(N \log(N))$ или за $O(N \log^2(N))$ в зависимости от реализации суффиксного массива (в случае использования хеширования в любом случае будет решение за $O(N \log(N))$). Такое решение получает полный балл (в том числе решение за $N \log^2(N)$ при должной аккуратности могло получить 100 баллов).

Разбор задачи «Монеты на дереве»

Постановка задачи

Дано дерево, и начальный набор, содержащее все вершины дерева, подвергается $n - 2$ удалениям. После каждого удаления необходимо определить кратчайшее расстояние между любыми двумя вершинами в наборе и количество таких пар. Запросы должны обрабатываться онлайн.

Решение оффлайн части

Сначала рассмотрим оффлайн часть. Легко обратить процесс операций, превратив его в добавление монет. Эта задача может быть легко поддержана с использованием дерева центроидов: листья центроидного дерева поддерживают самую дальнюю и вторую по дальности вершины от центроида. Для подсчёта количества пар это также можно сделать с помощью центроидного дерева после подсчета кратчайшего расстояния. Сложность алгоритма составляет $O(n \log n)$.

Онлайн часть для $n \leqslant 2 \cdot 10^5$

Для онлайн части заметим, что когда ответ равен ans, размер набора не превышает $\frac{2(n-1)}{ans}$. В процессе удаления ответ не монотонно убывает. Каждый раз, когда ответ изменяется, все точки добавляются обратно в текущий набор. Количество операций составляет $O(n \ln n)$. Когда количество пар падает до нуля, ответ изменяется. Количество пар также поддерживается с помощью центроидного дерева. Общая сложность составляет $O(n \ln n \log n)$.

Доказательство ограничения на размер множества

Рассмотрим окружности с радиусом $\frac{ans}{2}$ вокруг каждой точки набора на дереве. Все окружности не будут пересекаться друг с другом. Заменив каждое ребро дерева на два ребра, окружность вокруг точки будет содержать не менее ans рёбер. Поскольку всего имеется $2(n-1)$ рёбер, размер набора не может превышать $\frac{2(n-1)}{ans}$.

Онлайн часть для $n \leqslant 5 \cdot 10^5$

Рассмотрим поддержание кратчайшего пути, проходящего через каждый центроид, и количества путей такой длины для каждого центроида в центроидной декомпозиции. Затем ответ вычисляется единообразно. Рассмотрим непосредственное нахождение первой и второй ближайших точек к центроиду для формирования пути и подсчёта параметров пути. Если две точки происходят из одного поддерева, глобальный ответ точно будет меньше этой длины, поэтому некорректные пути не повлияют на ответ. Для каждого центроида заводится массив размером, равным максимальной глубине в компоненте связности, и поддерживаются два указателя для подсчёта ответа. Общая сложность составляет $O(n \log n)$.