Классы при механико-математическом факультете МГУ

Школа 54

Записи лекций Летней Школы по Программированию

С. С. Князевский

Красновидово

10 — 24 августа, 2024 год

Программа школы

1 Линейные алгоритмы

1.1 Префиксные суммы

Задача 1. Дан массив $\{a_0, a_1, \ldots, a_{n-1}\}$. На k-ом из m запросов даётся пара чисел (l_k, r_k), т. ч. $0 \leqslant l_k \leqslant r_k < n$; нужно найти сумму $a_{l_k} + a_{l_k+1} + + \ldots + a_{r_k}$.

Определение 1. *Массивом префиксных сумм* массива $\{a_0, a_1, \ldots, a_{n-1}\}$ называется массив (длины $n+1$) $\{p_0, p_1, \ldots, p_n\}$, для которого $p_i := \sum\limits_{j<i} a_j$. Иными словами, $p_0 = 0$, $p_1 = a_0$, $p_2 = a_0 + a_1$ и т. д.

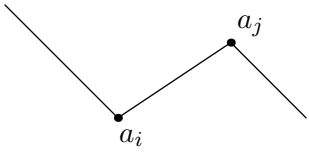
Нетрудно заметить, что $p_i = p_{i-1} + a_{i-1}$, поэтому массив префиксных сумм можно строить за линейное время. При этом, $\sum\limits_{i=l_k}^{r_k} a_i = p_{r_k+1} - p_{l_k}$, т. е. ответ на каждый запрос даём за $O(1)$, поэтому итоговая сложность составляет $O(n+m)$.

```
vector<int> p(n + 1); p[0] = 0;
for (int i = 1; i <= n; ++i)
    p[i] = p[i - 1] + a[i - 1]; // p_0 = 0, p_i = a_0 + ... + a_{i-1}.
int m;
cin >> m;
while (m--)
{
    int l, r;
    cin >> l >> r;
    cout << p[r + 1] - p[l] << '\n';
}
```

1.2 Поиск оптимальной пары

Задача 2. В массиве $\{a_0, a_1, \ldots, a_{n-1}\}$ найти пару (i, j) такую, что $i < j$ и значение $a_j - a_i$ максимально возможное.

Будем идти по массиву слева направо, перебирая правый элемент пары, а левый каждый раз выбирать наилучшим образом. Если правый элемент на данном шаге имеет индекс j, то наилучшим левым для него является $\min\{a_0, a_1, \ldots, a_{j-1}\}$, ведь разность тем больше, чем меньше вычитаемое. При этом,



на каждом новом шаге к рассмотрению добавляется ровно один элемент, поэтому для каждого правого элемента наилучший левый находим за $O(1)$, поэтому всё решение работает за $O(n)$.

```
int ibest = 0, jbest = 1;
int imin = 0;
for (int j = 2; j < n; ++j)
{
    if (a[j - 1] < a[imin]) imin = j - 1;
    if (a[j] - a[imin] > a[jbest] - a[ibest])
        ibest = imin, jbest = j;
}
cout << ibest << ' ' << jbest << '\n';
```

Задача 3. Даны массив $\{a_0, a_1, \ldots, a_{n-1}\}$ и натуральное число k. Нужно найти пару (i, j) такую, что $j - i \geqslant k$ и значение $a_i + a_j$ максимально возможное.

Здесь применяем ту же идею, что и в задаче о поиске оптимальной пары, но с некоторыми модификациями. Во-первых, начинаем перебирать элемент j не с начала, а с индекса k (ведь до этого мы очевидно не найдём подходящего индекса i). Во-вторых, лучшим элементом для j-го теперь является $\max\{a_0, a_1, \ldots, a_{j-k}\}$.

1.3 Суммы на отрезках

Задача 4. В массиве $\{a_0, a_1, \ldots, a_{n-1}\}$ нужно найти подотрезок $[a_l \ldots a_r]$, сумма на котором максимально возможная.

Сразу отметим, что если данный массив состоит из неотрицательных чисел, то ответом всегда будет являться весь массив; сложности возникают, если разрешены отрицательные числа. Однако они разрешимы, если уметь решать первые две задачи в этой лекции. Для решения можно предпосчитать массив префиксных сумм данного массива, а затем для него решить задачу о нахождении оптимальной пары.

Предположим теперь, что нам нужно найти лишь саму максимальную сумму, а не отрезок, на котором она достигается. У такой задачи есть очень изящное решение. Построим последовательность:

$$s_0 := 0, \quad s_i := \max\{s_{i-1} + a_{i-1}, 0\} \ (1 \leqslant i \leqslant n).$$

Фактически, мы просто накапливаем префиксную сумму, но если она становится отрицательной, мы ставим её в 0.

Утверждение 1. Максимальная сумма на подотрезке данного массива равна $\max\{s_0, \ldots, s_n\}$.

Доказательство. Точки i, в которых $s_i = 0$, назовём *критическими*. Заметим, что каждое s_i равно сумме на подотрезке от ближайшей слева критической точки до i (совпадение этих точек допускается в случае, если точка i сама является критической). Отметим, что для каждого i такая точка определена, т. к. $s_0 = 0$; обозначим её через i^*. Докажем, что максимальная сумма достигается на одном из таких отрезков. Рассмотрим произвольный отрезок $[a_l \ldots a_r]$. Сумма на отрезке $[a_{l^*} \ldots a_l]$ равна s_l и, как следствие, неотрицательна. Поэтому сумма на отрезке $[a_{l^*} \ldots a_r]$ не меньше, чем на $[a_l \ldots a_r]$, так что достаточно смотреть только на отрезки, начинающиеся в критических точках. ■

На самом деле, к описанному алгоритму тоже можно «прикрутить» нахождение отрезка, на котором достигается максимум. Искомый отрезок есть $[a_{j^*} \ldots a_j]$, где j — индекс максимального члена последовательности $\{s_i\}$, а j^* — ближайшая к нему слева критическая точка.

Задача 5. В массиве $\{a_0, a_1, \ldots, a_{n-1}\}$ неотрицательных чисел найти подотрезок $[a_l \ldots a_r]$, сумма на котором равна заданному значению k.

Будем искать отрезок следующим образом. Начнём с отрезка $[a_0]$, состоящим из одного элемента. На каждом шаге если сумма на отрезке меньше k, будем двигать правую границу отрезка на 1 вправо (при этом сумма на отрезке не уменьшится), если меньше — сдвинем левую границу на 1 вправо (при этом сумма на отрезке не увеличится). Из принципа дискретной непрерывности, мы либо найдём искомый отрезок, либо правый конец отрезка начнёт указывать на индекс n.

```
int i = 0, j, s = a[0];
for (j = 0; j < n; )
{
    if (s == k) break;
    if (s < k)
    {
        ++j;
        if (j < n) s += a[j];
    }
    else
    {
        s -= a[i];
        ++i;
    }
}
```

```
16
17  if (s == k)
18      cout << i << ' ' << j << '\n';
19  else
20      cout << "NO SOLUTION\n";
```

2 Корневые оптимизации

Иногда в задаче возникают ситуации, когда мы умеем решать её, когда для какой-то величины выполнено свойство, что она всегда либо меньше, либо больше, чем $\sqrt{n}$. Данный подход и называется корневой оптимизацией.

Примерами применения данной идеи могут служить: алгоритм проверки числа на простоту за $O(\sqrt{n})$ или факт, что если сумма неотрицательных чисел равна n, то различных среди них не более $\sqrt{n}$. Также можно вспомнить идею, что если $a \cdot b \leqslant n$, то одно из чисел a или b должно не превышать $\sqrt{n}$.

2.1 Корневая декомпозиция на массиве

Задача 1. Дан массив $\{a_0, a_1, \ldots, a_n\}$. Нужно обрабатывать на нём два типа запросов:

1. Найти сумму на отрезке $[a_l \ldots a_r]$;
2. Увеличить значение i-го элемента на x.

Разобьём массив на блоки размера $s = \lceil \sqrt{n} \rceil$ и в каждом блоке i предпосчитаем сумму b_i элементов в нём. Массив разбивается на блоки примерно так:

$$\underbrace{a_0, a_1, \ldots, a_{s-1}}_{b_0},\ \underbrace{a_s, a_{s+1}, \ldots, a_{2 \cdot s-1}}_{b_1},\ \ldots,\ \underbrace{a_{(s-1) \cdot s}, a_{(s-1) \cdot s+1}, \ldots, a_{n-1}}_{b_{s-1}}.$$

Последний блок может содержать меньше, чем s элеметов (если n не является полным квадратом), — это не существенно. Итак, на каждом блоке i мы знаем сумма на нём

$$b_i = \sum_{j=s \cdot i}^{\min\{n-1,(i+1) \cdot s-1\}} a_i.$$

Этот предпосчёт занял у нас $O(n)$ времени. Теперь чтобы посчитать сумму на отрезке $[a_l \ldots a_r]$, нужно посчитать за $O(\sqrt{n})$ суммы на перфиксе и суффиксе до ближайшего блока, а потом прибавить сумму чисел в нём.

```cpp
int n, m; // n - длина массива, m - число запросов
cin >> n >> m;
vector<int> a(n);
for (int i = 0; i < n; ++i)
    cin >> a[i];

// Предпосчёт
int s = (int)sqrt((ld)n) + 1;
vector<int> b(s, 0);
for (int i = 0; i < n; ++i)
    b[i / s] += a[i];

while (m--)
{
    int l, r;
    cin >> l >> r;
    int summ = 0;
    int i = l;
    while (i <= r)
    {
        if (!(i % s) && i + s - 1 <= r)
        {
            summ += b[i / s];
            i += s;
        }
        else
        {
            summ += a[i];
            ++i;
        }
    }
}
```

В блоках корневой декомпозиции можно хранить не только значения функций для подотрезка, а ещё и его отсортированную версию. Это бывает полезно при ответе на запросы вида «сколько элементов, меньших x, на отрезке» и используется в техниках `split-rebuild` и `split-merge`.

2.2 Split-rebuild

Задача 2. Пусть дан массив $\{a_0, a_1, \ldots, a_{n-1}\}$. К нему поступает M запросов, каждый одного из трёх видов:

1. Вставить элемент x на позицию i (т. е. слева от него должно оказаться i элементов);
2. Удалить элемент с позиции i;
3. Найти минимум на полуинтервале $[l; r)$.

2.2.1 Вставка

При вставке будем явно вставлять элемент в нужный блок. Если вставка происходит на границе блоков, то договоримся вставлять элемент в единственный существующий блок, если вставка происводится в самый конец или самое начало. Иначе вставляем в блок, найденный функцией `find_pos` (возвращает нужный блок и позицию в нём). Можно сделать иначе при вставке на концах — вставлять в новый блок, создавая его. Этот случай будет лучше работать, если у нас происходит много вставок подряд, а предыдущий — когда надо больше отвечать за запросы. Можно также принимать решение о вставке тем или иным способом случайно.

```
void insert_to_block(int pos, int elem)
{
    if (b.empty())
    {
        b.resize(1);
        b[0].push_back(elem);
        return;
    }
    pair<int, int> block_pos = find_pos(pos);
    if (block_pos.first == b.size())
    {
        b.back().push_back(elem);
        return;
    }
    insert(b[block_pos.first].begin() + block_pos.second, elem);
}
```

2.2.2 Удаление

При удалении будем явно удалять элемент из блока за размер блока. Если блок оказался пустым, то ничего с ним не будем делать пока что.

2.2.3 Функция на подотрезке

Запрос о вычислении функции обрабатываем, как в обычной корневой декомпозиции.

2.2.4 Перестраивание

Чтобы не допускать создания слишком большого числа маленьких блоков или разрастания отдельных блоков, раз в $\sqrt{n}$ операций будем заново полностью перестраивать структуру. Размер блока при этом тоже будет меняться. Также будем всегда поддерживать размер всей структуры.

```
void rebuild()
{
    int new_block_size = sqrt(b.size()) + 1;
    vector<int> all;
    for (vector<int> block : b)
        for (int elem : block)
            all.push_back(elem);

    b.clear();
    int new_blocks_cnt = (all.size() + new_block_size - 1) /
                         new_block_size;
    b.resize(new_blocks_cnt);
    for (int i = 0; i < all.size(); ++i)
    {
        b[i / new_block_size].push_back(all[i]);
        update(i); // обновить значение блока
        // в соответствие с задачей
    }
}
```

2.2.5 Как искать нужный блок

Т. к. теперь мы не можем гарантировать, что блок имеет фиксированный размер в каждый момент времени, то находить нужное место будем просто проходом по массиву блоков.

```
pair<int, int> find_pos(int pos)
{
    if (b.empty() || !pos)
        return {0, 0};
    int by_left = 0;
    int i = 0;
    while (i < b.size() && by_left < pos)
    {
        by_left += b[i].size();
        ++i;
```

```
11      }
12      by_left -= b[i].size();
13      --i;
14      return {i, pos - by_left};
15  }
```

2.3 Split-merge

2.3.1 Основная идея

В рамках техники `split-rebuild` мы регулярно перестраивали структуру данных. Оказывается, бывают ситуации, когда перестраивать структуру данных не так выгодно, как поддерживать её в сбалансированном состоянии.

Задача 3. Пусть дан массив $\{a_0, a_1, \ldots, a_{n-1}\}$. К нему поступает m запросов, каждый одного из четырёх видов:

1. Вставить элемент x на позицию i (т. е. слева от него должно оказаться i элементов);
2. Удалить элемент с позиции i;
3. Ответ на запрос на количество элементов, не меньших a на полуинтервале $[l; r)$; (`lower_bound`)
4. Массовые операции (например, переворот отрезка).

Заметим, что эту задачу мы уже могли решить с помощью `split-rebuild`. Для этого нам нужно было бы хранить для каждого блока его отсортированную версию. Если для каждого отсортированного элемента хранить его исходный индекс, то можно будет делать `split` за линейное время. Таким образом, у нас будет асимптотика $O(q\sqrt{n}\log n)$, ведь теперь для `rebuild` и `lower_bound` нам понадобится сортировка.

2.3.2 Склеиваем блоки

Теперь заметим, что мы можем склеить два соседних маленьких блока за $O(k)$, где k — размер блока, с помощью стандартного слияния. Тогда будем склеивать блоки, если существует пара соседних блоков, каждый из которых меньше, чем $\frac{k}{2}$. А резать блок будем, если его размер больше $2k$. Тогда блоков всегда будет $O\left(\frac{n}{k}\right)$, а размер блоков будет $O(k)$.

2.4 Корневая декомпозиция в задачах на графы

Определение 1. Назовём вершину *тяжёлой*, если она имеет более $\sqrt{|E|}$ соседей. Иначе, назовём вершину *тяжёлой*.

Лемма 1. В графе не более $2 \cdot \sqrt{|E|}$ тяжёлых вершин.

Доказательство. Пусть в графе более $2 \cdot \sqrt{|E|}$ тяжёлых вершин. Тогда числ о рёбер в графе больше, чем $\frac{2 \cdot \sqrt{|E|} \cdot \sqrt{|E|}}{2} = |E|$, противоречие. ■

Задача 4. Найти количество треугольников в графе за $O(|E| \sqrt{|E|})$.

Разобьём все вершины графа на лёгкие и тяжёлые. Заметим, что треугольников, образованных только тяжёлыми вершинами, всего $C^3_{\sqrt{E}} = = O(|E| \sqrt{|E|})$. Теперь рассмотрим треугольники, которые содержат в себе лёгкие вершины. В таком треугольнике точно будут два ребра, инцидентных лёгкой вершине. Сколько таких пар может быть? Всего таких рёбер $O(|E|)$, при этом для каждого ребра парными могут быть только $O(\sqrt{|E|})$ рёбер, в силу степени лёгкой вершины. Таким образом, число треугольников в графе равно $O(|E| \sqrt{|E|})$. Каким алгоритмом их искать? Можно явно провести процесс, описанный выше, но это не самое приятное в реализации решение этой задачи.

Можно переориентировать рёбра от вершин с меньшей степенью к вершинам с большей. Теперь верно следующее.

Лемма 2. Из каждой вершины выходит не более $O(\sqrt{|E|})$ рёбер.

Доказательство. Степень лёгких вершин $O(\sqrt{|E|})$, а из тяжёлых вершин рёбра идут только в тяжёлые, которых всего $O(\sqrt{|E|})$. ■

Теперь для каждой вершины пометим её соседей, после чего запустим поиск путей длины 2 и будем фиксировать треугольник при нахождении пометки. Для каждого первого ребра пути мы посмотрим на $O(\sqrt{|E|})$ рёбер, поэтому итоговая сложность алгоритма $O(|E| \sqrt{|E|})$.

2.5 Корневая декомпозиция на строках

Данные идеи очень полезны в задачах, где есть ограничение на суммарный размер строк. Обозначим это ограничение за $\sum |s|$.

Определение 2. Назовём строку s *длинной*, если $|s| \geqslant \sqrt{\sum |s|}$. Иначе будем называть строку s *короткой*.

Лемма 3. Существует не более $\sqrt{\sum |s|}$ длинных строк.

Доказательство. Пусть существует более $\sqrt{\sum |s|}$ длинных строк. Тогда их суммарная длина больше $\sqrt{\sum |s|} \cdot \sqrt{\sum |s|} = \sum |s|$, противоречие. ■

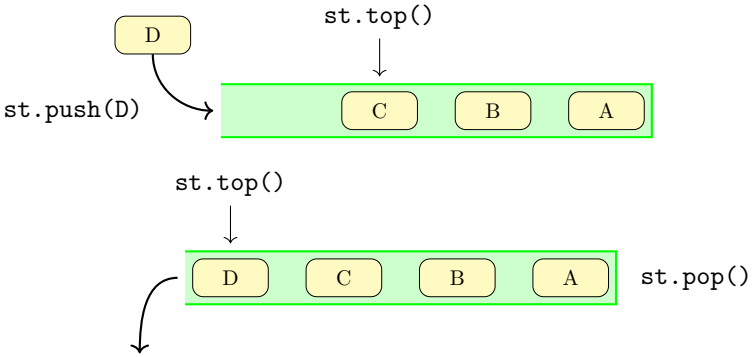
Лемма 4. Количество различных длин строк не более, чем $O(\sqrt{\sum |s|})$.

Доказательство. Пусть количество различных длин равно x. Тогда минимальная возможная сумма длин — это сумма чисел от 1 до x, равная $\frac{x(x+1)}{2} < \sum |s|$, отсюда $x = O(\sqrt{\sum |s|})$. ■

3 Структуры данных в линейных решениях

3.1 Стек и его применения в задачах

Определение 1. *Стек* — структура данных, представляющая из себя упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого *вершиной стека*.



Задача 1. Для каждого элемента массива $\{a_1, a_2, \ldots, a_n\}$ найти ближайший справа элемент, меньший него.

Если решать «в лоб», то максимальное количество операций равно

$$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = O(n^2),$$

достигается при отсортированном по возрастанию массиве.

Для линейного решения предварительно добавим в массив граничные элементы $a_0 := -\infty$, $a_{n+1} := -\infty$ и создадим стек, положив в него индекс 0. Будем идти по массиву $\{a_1, a_2, \ldots, a_{n+1}\}$ слева направо и сравнивать каждый элемент массива a_i с элементом массива, доступным по верхнему индексу стека. Пока a_i меньше верхнего элемента стека, будем удалять этот верхний элемент и записывать i в качестве ответа для него. Затем добавим i в стек.

Обоснуем корректность этого алгоритма. Сначала заметим, что стек никогда не остаётся пустым и каждый элемент хоть раз будет в него добавлен (в силу того, что $a_0 = -\infty$). Также, для каждого элемента будет записан ответ (в силу того, что $a_{n+1} = -\infty$). Записанный ответ является правильным, т. к. для каждого числа записывается первое, меньшее него.

```
vector<int> ans(n + 2);
stack<int> st; st.push(0);
for (int i = 1; i < n + 2; ++i)
{
    while (a[st.top()] > a[i])
        ans[st.top()] = i, st.pop();
    st.push(i);
}
```

Задача 2 (Гистограмма)**.** Дан массив $\{h_1, h_2, \ldots, h_n\}$, представленный в виде гистограммы. Требуется найти наибольшую площадь прямоугольника, вписанного в эту гистограмму.

Заметим, что искомый прямоугольник имеет высоту, совпадающую с высотой одного из столбцов. Действительно, если это не так, то можно увеличить его площадь, подняв до нижнего столбца в гистограмме. Будем перебирать столбцы и для каждой из них находить левую и правую границы с помощью предыдущей задачи. Для каждого столбца это можно сделать заранее и ответами заполнить массивы $\{l_1, l_2, \ldots, l_n\}$ и $\{r_1, r_2, \ldots, r_n\}$. Тогда площадь прямоугольника, покрывающего i-й столбец (высотой h_i) равна $S_i = h_i(r_i - l_i - 1)$.

Задача 3. В массиве $\{\overline{a_1, \underbrace{a_2, \ldots, a_k}, a_{k+1}}, \ldots, a_n\}$ найти минимум в скользящем окне шириной k.

Решим задачу о нахождении ближайшего справа элемента меньше текущего и ответы запишем в массив $\{b_0, b_1, \ldots, b_{n+1}\}$. Положим $i = 1$ (указывает на первый элемент первого окна), а затем будем делать замену $i \mapsto b_i$, пока b_i не выходит за границы окна. В конце i будет указывать на минимальный элемент в этом окне, т. к. a_i меньше всех элементов окна, стоящих левее него, а первый элемент правее и меньше него уже не попадает в окно. При переходе к новому окну если текущее значение i в него не попадает, то ставим i на позицию первого элемента нового окна.

Данный алгоритм работает за линейное время, т. к. $b_i > i$ для каждого индекса i. Таким образом, указатель на минимальный элемент окна не убывает. Всего элементов в массиве n, значит, изменений указателя не более n. Худший случай — отсортированный по убыванию массив, на котором каждый раз мы двигаемся ровно на 1 шаг.

```cpp
int i = 1;
vector<int> ans(n - k + 2);
for (int j = 1; j <= n - k + 1; ++j)
{
    if (i < j) i = j;
    while (b[i] < j + k) i = b[i];
    ans[j] = i;
}
```

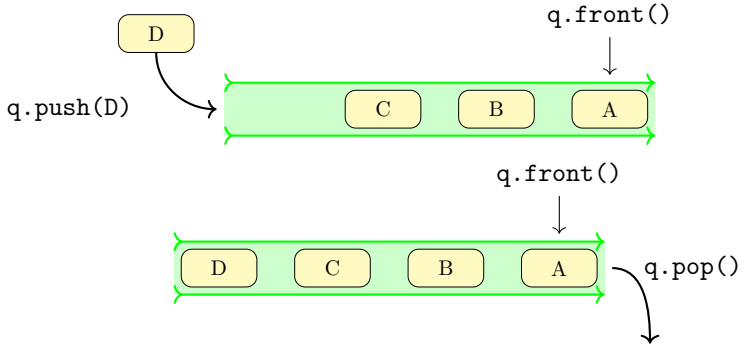
Определение 2. Скобочная последовательность s называется *правильной*, если s пуста или $s = (s')$, где s' — правильная скобочная последовательность, или $s = s_1 s_2$, где s_1 и s_2 — правильные скобочные последовательности.

Задача 4. По данной скобочной последовательности проверить, является ли она правильной.

Заведём стек. Последовательно перебираем символы строки. Если встречаем открывающую скобку, добавляем её в стек, а если увидели закрывающую скобку, то сверху стека должна быть открывающая, парная текущей закрывающей. Тогда удалим эту открывающую скобку из стека. Если по окончании стек не пустой или во время работы не нашли парную для какой-то закрывающей скобки, данная скобочная последовательность не является правильной; иначе — является.

3.2 Очередь

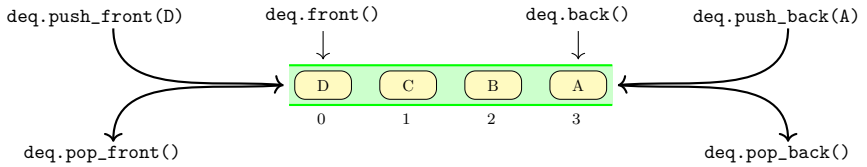
Определение 3. *Очередь* — это структура данных, добавление и удаление элементов в которой происходит так, что первым из очереди удаляется элемент, который был помещен туда первым. У очереди имеется *хвост*, куда добавляются элементы, и *голова*, откуда они удаляются.



Одним из основных применений очереди является реализация поиска в ширину, про это будет рассказано позже.

3.3 Дек и второе решение задачи о минимуме в окне

Определение 4. *Дек* — структура данных, представляющая из себя список элементов, в которой добавление новых элементов и удаление существующих производится с обоих концов.



Отметим, что на дек бывает удобно смотреть как на двустороннюю очередь с индексацией.

Предложим другое решение задачи о поиске минимума в скользящем окне в массиве $\{a_0, a_1, \ldots, a_{n-1}\}$ (в этот раз нам удобнее вести индексацию с 0). Назовём элемент *перспективным* для данного окна, если в этом окне нет элементов правее и не больше него. Это условие необходимо для того, чтобы какой-то элемент был минимумом в данном окне, поэтому этот минимум принадлежит множеству перспективных элементов. Заметим, что последовательность перспективных элементов неубывающая, а потому минимумом является первый элемент этой последовательности. Искать его будем так: заведём дек и сначала положим в него 0 (указывает на первый элемент первого окна), а каждый следующий элемент будем добавлять, предварительно удалив из дека все элементы, большие этого следующего. Для перехода к новому окну удалим из дека первый элемент предыдущего окна, если он там есть (при этом он всегда будет в начале дека), и добавим последний элемент нового указанным способом.

```
deque<int> deq;
for (int i = 0; i < n; ++i)
{
    if (i >= k && deq.front() == i - k) deq.pop_front();
    while (deq.size() && a[deq.back()] >= a[i])
        deq.pop_back();
    deq.push_back(i);
    if (i >= k - 1)
        ans.push_back(deq.front());
}
```

Для каждого из вышеперечисленных контейнеров определены методы `size()`, возвращающий количество его элементов, и `empty()`, проверяющий его на пустоту.

4 Дерево отрезков

Пусть дан массив $\{a_0, \ldots, a_{n-1}\}$ и функция f, т. ч.

$$f\big(f([a_i \ldots a_j]), f([a_{j+1} \ldots a_k])\big) = f([a_i \ldots a_k])$$

для любых i, j, k, т. ч. $0 \leqslant i \leqslant j < k < n$.

Мы научимся делать следующее:

1. Для любых l и r $(0 \leqslant l \leqslant r < n)$ возвращать значение $f([a_l \ldots a_r])$ за $O(\log n)$;
2. Для любых i $(0 \leqslant i < n)$ и x заменять значение элемента a_i на x за $O(\log n)$;
3. Для любых l, r $(0 \leqslant l \leqslant r < n)$ и x заменять значение каждого элемента на отрезке $[a_l \ldots a_r]$ на x за $O(\log n)$. (Обобщение п. 2)

Пример 1. В качестве f можно брать, например, сумму, max или min.

4.1 Основная идея

Для этого мы рассмотрим структуру данных, называемую *деревом отрезков*. Она представляет собой бинарное дерево, в котором в листьях хранятся элементы массива, а в каждой из остальных вершин — результат вычисления функции f от её сыновей.

Для вычисления функции на отрезке $[a_l \ldots a_r]$ мы будем спускаться по дереву и рассматривать три случая:

1. Отрезок, значение функции f на котором хранится в текущей вершине, не пересекается с $[a_l \ldots a_r]$. Тогда возвращаем нейтральный элемент (такой элемент e, что $f(x, e) = f(e, x) = x$ для любого x).
2. Отрезок, значение функции f на котором хранится в текущей вершине, содержится (в нестрогом смысле) в $[a_l \ldots a_r]$. Тогда возвращаем значение в вершине.
3. Отрезок, значение функции f на котором хранится в текущей вершине, пересекается с $[a_l \ldots a_r]$. Тогда опускаемся на один уровень ниже в её сыновей.

Пример 2. Нейтральным элементом для суммы является 0, для максимума $-\infty$.

Теорема 1. Высота дерева отрезков равна $\lceil \log_2 n \rceil$.

Доказательство. На последнем уровне в дереве должно находиться (не более) n вершин, вершин на i-ом слое (кроме, возможно, последнего) ровно 2^i (начиная с нуля). Наименьшее i, при котором $2^i \geqslant n$, равно $\lceil \log_2 n \rceil$. ■

Следствие 1. Дерево отрезков хранит не более $4n$ вершин.

Доказательство. Первый уровень содержит одну вершину (корень), второй — две, третий — четыре и так до n. Суммируя, получаем $1 + 2 + \ldots + 2^{\lceil \log_2 n \rceil} = 2^{\lceil \log_2 n \rceil + 1} < 4n$. ■

Следствие 2. Время одной операции в дереве отрезков $O(\log n)$.

Доказательство. Мы посещаем не более двух вершин на каждом слое, отсюда число операций составляет не более $2\lceil \log_2 n \rceil$. ■

Приступим к описанию реализации. Хранить дерево будем как массив, элементы которого отвечают за результат для текущей вершины в дереве. Из вершины с индексом v в данном массиве виден левый сын по индексу $2v$, а правый — по индексу $2v + 1$. Есть ещё некоторые приёмы, которые существенно упрощают написание:

1. Определиться заранее с нейтральным элементом;
2. Написать один раз функцию `combine`, которая комбинирует результаты левого и правого сыновей;
3. В качестве границ брать полуинтервалы вида $[l; r)$, а не отрезки.

Обозначения: v — некоторая вершина дерева отрезков, которая хранит результат вычисления функции f на полуинтервале $[t_l; t_r)$. Она является листом тогда и только тогда, когда $t_l + 1 = t_r$. Запросы будут подаваться на полуинтервалах $[q_l; q_r)$. Тогда отрезок, за который отвечает левый сын вершины v, есть $\left[t_l; \frac{t_l + t_r}{2}\right)$, а правый — $\left[\frac{t_l + t_r}{2}; t_r\right)$.

4.2 Построение

```
1  void build(int v, int tl, int tr)
2  {
3      // Если v - лист, положить в него элемент массива
4      if (tl + 1 == tr)
5          t[v] = a[tl];
6      // Иначе положить в вершину комбинацию значений её сыновей
7      else
8      {
9          int tm = (tl + tr) / 2;
```

```
        build(2 * v, tl, tm);
        build(2 * v + 1, tm, tr);
        t[v] = combine(t[2 * v], t[2 * v + 1]);
    }
}
```

4.3 Модификация

Когда изменяется элемент массива, нужно изменить соответствующие вершины в дереве: нужно обновить лист, а также пересчитать значения, которые зависели от этого элемента. Такие вершины лежат по одной на каждом уровне от корня до изменяемого листа. Значит, их количество не более $\log_2 n$.

Находясь в вершине, нам надо спуститься в того сына, который отвечает за отрезок, в котором произойдёт изменение.

```
void upd(int v, int tl, int tr, int pos, int val)
{
    if (tl + 1 == tr)
    {
        t[v] = val;
        return;
    }

    int tm = (tl + tr) / 2;
    if (pos < tm)
        upd(2 * v, tl, tm, pos, val);
    else
        upd(2 * v + 1, tm, tr, pos, val);
    t[v] = combine(t[2 * v], t[2 * v + 1]);
}
```

4.4 Получение результата

```
int get(int v, int tl, int tr, int ql, int qr)
{
    if (qr <= tl || tr <= ql)
        return NEUTRAL; // [t_l; t_q] не пересекается с [q_l; q_r]
    if (ql <= tl && tr <= qr)
        return t[v]; // [t_l; t_q] содержится в [q_l; q_r]

    // Иначе комбинируем результат сыновей
```

```
    int tm = (tl + tr) / 2;
    return combine(get(2 * v, tl, tm, ql, qr), \
                   get(2 * v + 1, tm, tr, ql, qr));
}
```

4.5 Массовые операции

Теперь пусть запрос модификации представляет собой прибавление ко всем числам на некотором подотрезке $[a_l \dots a_r]$ некоторого числа x, а запрос чтения — считывание некоторого числа a_i.

Чтобы обрабатывать запрос прибавления эффективно, будем хранить в каждой вершине дерева отрезков, сколько нужно прибавить к каждому числу этого отрезка. Тем самым мы сможем обрабатывать запрос прибавления на любом подотрезке эффективно за $O(\log n)$ вместо того, чтобы на каждом запросе менять все $O(n)$ значений.

Чтобы обработать запрос чтения значения a_i, достаточно спуститься по дереву, просуммировав все встреченные по пути значения, записанные в вершинах дерева. Сложность этого решения составляет $O(\log n)$ из-за высоты дерева.

```
void upd(int v, int tl, int tr, int ql, int qr, int x)
{
    if (ql >= qr)
        return;
    if (ql == tl && qr == tr)
        t[v] += x;
    else
    {
        int tm = (tl + tr) / 2;
        upd(2 * v, tl, tm, ql, min(qr, tm), x);
        upd(2 * v + 1, tm, tr, max(ql, tm), qr, x);
    }
}

int get(int v, int tl, int tr, int pos)
{
    if (tl + 1 == tr)
        return t[v];
    int tm = (tl + tr) / 2;
    if (pos < tm)
        return t[v] + get(2 * v, tl, tm, pos);
    else
```

```
23          return t[v] + get(2 * v + 1, tm, tr, pos);
24  }
```

Попробуем обобщить эту идею для произвольной операции и посмотрим, в какие ограничения мы упираемся. Пусть $\otimes$ — некоторая операция. Рассмотрим, что будет происходить со значением a_i при запросах x и y, которые его изменяли.

Спускаясь по дереву от корня до листа, мы пересчитываем значение не в том порядке, как мы их должны применить, следуя запросам, так что $(a_i \otimes x) \otimes y = a_i \otimes (x \otimes y)$, т. е. операция $\otimes$ должна быть ассоциативна.

Также значение a_i не должно зависеть от порядка запросов над ним, так что $a_i \otimes x \otimes y = a_i \otimes y \otimes x$ — это коммутативность.

4.6 Некоммутативные операции и присвоение на отрезке

Научимся работать с некоммутативными операциями на примере присвоения, ведь его можно рассматривать как операцию $x \odot y := y$.

Примечание. Заметим, что присвоение является ассоциативной операцией: $(x \odot y) \odot z = z$, $x \odot (y \odot z) = x \odot z = z$.

Рассмотрим задачу, в которой запрос чтения — получение значения массива a_i, а модификации есть присвоение всем элементам некоторого отрезка $[a_l \dots a_r]$ некоторого значения p. При этом будем говорить, что мы красим отрезок $[a_l \dots a_r]$ в цвет p.

Чтобы делать модификацию на целом отрезке, разобьём его на набор подотрезков, каждый из которых покрывается какой-то вершиной дерева. Разбиение мы делаем спуском, как и до этого. Мы будем красить не каждый элемент массива на интересующем нас отрезке, а только вершины полученного разбиения. То же самое мы делали, когда решали задачу о прибавлении на отрезке.

Итак, после выполнения запроса модификации дерево отрезков становится, вообще говоря, неактуальным — в нём остались недовыполненными некоторые модификации.

Теперь предположим, что после покраски отрезка $[a_l \dots a_r]$ в какой-то цвет нам пришёл запрос модификации какого-то его подотрезка $[a_{l'} \dots a_{r'}]$ в другой цвет. Мы хотим покрасить вершины разбиения этого подотрезка (которые являются, возможно, непрямыми, потомками ранее покрашенных вершин). Проблема в том, что до покраски этих вершин мы должны разобраться с изначальными вершинами разбиения, при этом утратится информация о покраске для вершин из нашего отрезка, не входящих в $[a_{l'} \dots a_{r'}]$.

Выход в том, чтобы произвести *проталкивание* информации из корня, т. е. если корень поддерева был покрашен в какой-либо цвет, то покрасить в этот цвет его правого и левого сына, а из корня эту отметку убрать. После этого мы можем красить сыновей корня, не теряя никакой важной информации.

Асимптотика такого решения есть $O(\log n)$, что доказывается аналогично следствию 2.

Для реализации нам нужно написать дополнительную функцию, которая будет производить проталкивание информации из вершины в её сыновей, и вызывать эту функцию в самом начале обработки запросов (но не из листьев).

```
void push(int v)
{
    if (t[v] >= 0)
    {
        t[2 * v] = t[2 * v + 1] = t[v];
        t[v] = -1; // обозначение отсутствия изменения
    }
}

void upd(int v, int tl, int tr, int ql, int qr, int p)
{
    if (ql >= qr)
        return;
    if (ql == tl && qr == tr)
        t[v] = p;
    else
    {
        push(v);
        int tm = (tl + tr) / 2;
        upd(2 * v, tl, tm, ql, min(qr, tm), p);
        upd(2 * v + 1, tm, tr, max(ql, tm), qr, p);
    }
}

int get(int v, int tl, int tr, int pos)
{
    if (tl + 1 == tr)
        return t[v];
    push(v);
    int tm = (tl + tr) / 2;
```

```
31      if (pos < tm)
32          return get(2 * v, tl, tm, pos);
33      else
34          return get(2 * v + 1, tm, tr, pos);
35  }
```

Примечание. Функцию `get` можно было реализовывать и по-другому: не делать в ней запаздывающих обновлений, а сразу возвращать ответ, как только мы попадаем в вершину дерева, целиком покрашенную в тот или иной цвет.

5 Введение в графы. Обход в ширину

Будем рассматривать граф G с множеством вершин $V := \{0, 1, \ldots, n - 1\}$ и множеством рёбер E.

5.1 Способы хранения графов

Есть несколько способов хранения графов, каждый из которых удобен в разных ситуациях.

1. **Список рёбер**. Название говорит само за себя: заводится массив, в котором хранятся рёбра в виде пар инцидентных вершин. Применяется, например, в алгоритме Краскала (см. лекцию по СНМ).

2. **Матрица смежности**. Заведём двумерный массив a размера $n \times n$, заполненный следующим образом:

$$a_{ij} = \begin{cases} 1, & \text{если } (i, j) \in E, \\ 0, & \text{иначе.} \end{cases}$$

   Отметим, что для неориентированного графа матрица смежности является симметрической.

3. **Список смежности**. Заведём двумерный массив a размера n, в котором по индексу i хранится массив вершин, инцидентных i. Этот способ обычно удобнее всего, т. к. 1) удобнее всего получать соседей для каждой вершины; 2) суммарный размер $O(m)$, а не $O(n^2)$, что полезно, когда $m \sim n$.

5.2 Идея поиска в ширину

При обходе в ширину мы идём по вершинам в порядке, в котором они бы сгорали, если поджечь начальную (каждую секунду огонь распространяется на соседние вершины). Моделируем данный процесс, пока не сгорит весь граф. Важно помнить, что мы не поджигаем уже горящие вершины. Т. к. огонь распространяется равномерно по кратчайшим путям, секунда, на которой сгорела i-я вершина — это расстояние до неё.

5.3 Реализация

```
// Список смежности графа G размера n
vector<vector<int>> adj;
// Векторы для хранения расстояний до каждой вершины
// и её предков
vector<int> dist, parent;

void bfs(int start)
{
    queue<int> q; // Очередь для хранения горящих вершин
    dist.assign(n, INF);
    parent.assign(n, -1);
    q.push(start);
    dist[start] = 0;
    while (!q.empty())
    {
        int v = q.front();
        q.pop();
        for (auto u : adj[v])
        {
            if (dist[u] == INF)
            {
                dist[u] = dist[v] + 1;
                parent[u] = v;
                q.push(u);
            }
        }
    }
}
```

Множество пройденных ребёр (v, p_v) образует дерево (*дерево обхода*).

6 Система непересекающихся множеств

Задача 1. Дан неориентированный граф. По двум вершинам нужно понимать, находятся ли они в одной компоненте связности. Имеется два типа запросов:

1. Объединение — построить ребро между двумя вершинами;
2. Связность — по двум вершинам проверить, находятся ли они в одной компоненте.

Наивное решение (с помощью обхода в ширину) имеет асимптотическую сложность $O((n+m)\cdot q)$. Это не будет проходить по времени при вполне разумных ограничениях по типу $n, q \leqslant 10^5$.

Ещё можно пробовать красить компоненты в отдельные цвета. Проблема в том, что тогда при объединении двух компонент, нужно хотя бы одну из них полностью перекрасить (в другой цвет), и каждое обновление будет происходить за $O(n)$.

6.1 Основная идея

Для каждой компоненты связности создаём ориентированное остовное дерево, в котором каждое ребро направлено от листьев к корню. Тогда для любых двух вершин будем подниматься до корней, и если они совпали, то они находятся в одном множестве; иначе — нет.

Для оптимизации при подвешивании деревьев друг к другу (объединении множеств) будем делать так: корень меньшего дерева будем подвешивать к корню большего. Тогда мы будем получать самые низкие деревья, чтобы меньше по ним ходить.

Теорема 1. Высота каждого дерева растёт как $O(\log n)$.

Доказательство. Проведём индукцию по размеру деревьев v.

База ($v = 1$). На графе без рёбер размер деревьев равен $1 = \log_2 0$.

Шаг ($v = v_1 + v_2$). Пусть имеем два дерева с v_1 и v_2 вершинами соответственно. Не ограничивая общности, считаем, что $v_1 \leqslant v_2$. Обозначим через h_1 и h_2 высоты соответствующих деревьев, а через $h_{1\cup 2}$ обозначим высоту объединения деревьев. По предположению индукции $h_i \leqslant \lceil \log_2 v_i \rceil$ $(i = 1, 2)$. Рассмотрим два случая:

1. $h_1 \neq h_2$. Тогда $h_{1\cup 2} = \max\{h_1, h_2\}$. Отсюда,

$$h_{1\cup 2} = \max\{h_1, h_2\} \leqslant \max\{\lceil \log_2 v_1 \rceil, \lceil \log_2 v_2 \rceil\} \leqslant \lceil \log_2 (v_1 + v_2) \rceil,$$

где последнее неравенство выполнено в силу монотонности функции логарифма.

2. $h_1 = h_2$ ($=: h$). Тогда $h_{1\cup 2} = h + 1$, причём по предположению индукции $h \leqslant \min\{\lceil \log_2 v_1 \rceil, \lceil \log_2 v_2 \rceil\} = \lceil \log_2 v_1 \rceil$ (опять же в силу монотонности). Теперь

$$h_{1\cup 2} = h + 1 \leqslant \lceil \log_2 v_1 \rceil + 1 = \lceil \log_2 (2 \cdot v_1) \rceil \leqslant \lceil \log_2 (v_1 + v_2) \rceil.$$

Таким образом, высота любого дерева в СНМ есть $O(\log n)$. ■

Следствие 1. Время выполнения одной операции в СНМ есть $O(\log n)$.

Доказательство. Для проверки мы поднимаемся от вершин к их корням, делая не более $2 \log_2 n = O(\log n)$ шагов. ■

6.2 Реализация

```
vector<int> link; // ссылка на корень, по умолчанию храним −1
vector<int> h_set; // высота поддерева вершины, сначала равна 0

// По вершине находит корень её дерева
int find(int v)
{
    if (link[v] == -1)
        return v;
    return find(link[v]);
}

// По 2-м вершинам объединяет деревья, в которых они находятся
void unite(int u, int v)
{
    u = find(u), v = find(v);

    if (u == v) return;

    if (h_set[u] > h_set[v])
        swap(u, v);

    link[u] = v;
    h_set[v] += (h_set[u] == h_set[v]);
}

bool check(int u, int v)
{
    u = find(u), v = find(v);
```

```
29
30      return u == v;
31  }
```

Как уже было доказано выше, каждая операция в CHM работает за $O(\log n)$. Этот алгоритм можно ещё улучшить. Если имеем поддерево $i \leftarrow$ $\leftarrow j \leftarrow k$, то можно его переподвесить $i \leftarrow k$. Для этого в функции `find` нахождения корня вместо `return find(link[v]);` нужно писать `return (link[v] = find(link[v]));`, тогда все вершины переподвешиваются к одному корню. Однако так нельзя делать в задачах, где нужно помнить, в какие моменты времени мы объединяли деревья, т. к. при таком подходе эта информация теряется.

Асимптотика такого решения составляет $O(\alpha(n, m))$ (n — число вершин в графе, m — число запросов), где α — обратная функция Аккермана. Отметим, что при любых значениях n и m, возникающих в прикладных задачах, $\alpha(n, m) \leqslant 4$, так что можно считать, что функция `find` выполняется за константное время.

6.3 Поиск минимального остова

Определение 1. *Остовом* графа называется поддерево, содержащее все его вершины. *Минимальным остовом* ориентированного графа называется его остов с минимальным суммарным весом рёбер.

Алгоритм Краскала. Заведём список рёбер графа G, отсортируем его по увеличению веса и будем идти по этому списку. Для каждого ребра посмотрим, находятся ли его вершины в одном множестве. Если нет — связываем эти множества (проводим ребро в остове). Асимптотика составляет $O(m \log m + m\alpha(n, m))$.

7 Обход в глубину

7.1 Идея обхода в глубину

Из каждой вершины идём в любую свободную и красим её как посещённую. Идём так, пока есть непосещённые вершины. Если «зашли в тупик» (все соседние вершины покрашены), возвращаемся назад. Делаем так, пока все вершины не будут покрашенными. Асимптотика $O(n + m)$.

Пример простейшей задачи, которую можно решить с помощью обхода в глубину — поиск числа компонент связности в графе. Для этого можно из каждой непосещённой вершины запускать обход в глубину, пока

все вершины не будут посещены. Количество запусков, которое пришлось сделать, и есть число компонент.

7.2 Реализация

```
// Вектор размера n для раскраски посещённых вершин
vector<int> used;
// Список смежности графа G размера n
vector<vector<int>> adj;

void dfs(int v)
{
    used[v] = 1; // отмечаем текущую вершину
    // обрабатываем текущую вершину, если нужно
    for (auto u : adj[v])
        if (!visited[u]) dfs(u);
}
```

7.3 Поиск циклов и покраска в три цвета

Задача 1. Найти цикл в данном графе.

Ребро к посещённой вершине, не являющейся предком текущей при обходе в глубину, будем называть *обратным*.

Ясно, что в данной компоненте связности графа есть цикл если и только если при обходе из какой-то вершины из этой компоненты мы нашли обратное ребро. Для восстановления ответа (т. е. для нахождения самого цикла) можно хранить массив предков и при нахождении обратного ребра с какой-то вершиной начинать подниматься по нему до того момента, как опять придём в эту же вершину.

Есть проблема — такой способ не работает на ориентированных графах. В качестве простого контрпримера можно рассмотреть цикл на трёх вершинах с одним развёрнутым ребром. При рассмотрении этого примера видно, как решить проблему. Нам нужно отделять вершины, из которых мы уже вышли, от тех, которые ещё в обработке, для этого можно завести отдельный цвет.

7.4 Ещё задачи

Определение 1. Граф называется *двудольным*, если его вершины можно покрасить в два цвета так, чтобы ни одна пара вершин одного цвета не была соединена ребром.

Задача 2. Проверить данный граф на двудольность.

Будем красить вершины в три цвета — непосещённые и отдельно две доли. Если граф не связный, нужно проверить на двудольность каждую компоненту. Иначе начинаем обход из любой вершины и красим следующую вершину не в тот цвет, в который покрасили предыдущую.

Задача 3. У профессора записаны все пары студентов, которые списывали или давали списывать. Требуется определить, сможет ли он разделить студентов на две группы так, чтобы любой обмен записками осуществлялся от студента одной группы студенту другой группы.

По сути, задача заключается в проверке графа на двудольность с данными долями.

Задача 4. Дан связный граф, в котором n вершин и m ребер, требуется удалить наименьшее количество ребер так, чтобы получившийся граф стал деревом.

8 Перебор и динамика по подотрезкам, поддеревьям, подмножествам, подмаскам, ...

8.1 Линейная динамика

Напомним, что мы называем *линейным динамическим программированием* и рассмотрим пример задачи, которую оно решает.

Задача 1. Есть дорожка из n клеток (нумеруем с 0), Марио стоит в нулевой. Известно, что некоторые клетки залиты лавой, на них наступать нельзя. Они указаны в массиве отметок $\{a_0, a_1, \ldots, a_{n-1}\}$ ($a_i = 1$, если i-я клетка залита лавой, иначе $a_i = 0$). Марио умеет прыгать на одну или две клетки вперёд. Сколькими способами он может попасть из нулевой клетки в $(n-1)$-ю?

Пусть d_i — число способов попасть из нулевой клетки в i-ю ($0 \leqslant i < n$). В качестве базы возьмём $d_0 = 1$ (попасть из нулевой клетки в себя можно только одним способом — пустым путём) и $d_1 = 1 - a_1$. Здесь мы учитываем, что клетка с номером 1 может быть залита лавой. Тогда имеем рекуррентную формулу

$$d_i = (1 - a_{i-1}) \cdot d_{i-1} + (1 - a_{i-2}) \cdot d_{i-2}.$$

Результат хранится в d_{n-1}, и его вычисление заняло $O(n)$ времени.

8.2 Динамика по подотрезкам

Задача 2. Дана строка s длины n. Найти количество её подстрок, являющихся палиндромами.

Заведём двумерный массив d размера $(n+1) \times (n+1)$; пусть $d_{l,r} = 1$, если подстрока $[s_l \dots s_r)$ является палиндромом, иначе $d_{l,r} = 0$. Тогда условия $d_{i,i} = 1$ $(0 \leqslant i \leqslant n)$ и $d_{i,i+1} = 1$ $(0 \leqslant i < n)$ можно взять за базу (все подстроки длины 0 и 1 являются палиндромами). Тогда при $r > l+1$ имеем

$$d_{l,r} = \begin{cases} 1, & \text{если } s_l = s_{r-1} \text{ и } d_{l+1,r-1} = 1, \\ 0, & \text{иначе.} \end{cases}$$

Можем вести цикл по левым границам и длинам отрезков, заполняя наш массив за $O(n^2)$ по указанной формуле.

8.3 Динамика по поддеревьям

Задача 3. Дано дерево G. Найти сумму всех путей в нём.

Наивное решение: перебираем вершины и любым обходом ищем путь из неё во все остальные. Асимптотика составляет $O(n \cdot (n+m))$, где $m = n-1$ (т. к. G — дерево), т. е. $O(n^2)$.

Подвесим дерево за какую-то вершину. Обозначим через d_v число вершин в поддереве v. Тогда число путей, содержащих ребро (u, v) (где v — непосредственный сын u при подвешивании) равно $d_v \cdot (n - d_v)$. Тогда ответом будет

$$\sum_{(u,v) \in E} d_v \cdot (n - d_v).$$

```
vector<vector<int>> adj; // список смежности дерева G
vector<int> d(n, 1); // пока в поддереве u одна вершина - u
vector<int> p; // храним предков
vector<int> cnt_sons; // храним число сыновей

queue<int> q;
q.push(0);
while (!q.empty())
{
    int u = q.front(); q.pop();
    for (int v : adj[u])
        if (v != p[u])
        {
```

```
            p[v] = u;
            ++cnt_sons[u];
            q.push(v);
        }
}

for (int u = 0; u < n; ++u)
    if (!cnt_sons[u]) // тогда u - лист
        q.push(u);

int ans = 0;
while (!q.empty())
{
    int v = q.front(); q.pop();
    u = p[v];
    ans += d[v] * (n - d[v]);
    d[u] += d[v];
    if (!(--cnt_sons[u]))
        q.push(u);
}
```

8.4 Динамика по подмножествам или подмаскам

Определение 1. Путь в графе G называется *гамильтоновым*, если он проходит через все вершины ровно по одному разу.

Задача 4. В данном взвешенном графе G найти минимальный по весу гамильтонов путь (если он существует).

Обозначим через $w_{u,v}$ вес ребра (u, v) (удобно считать $w_{u,v} = +\infty$, если ребра (u, v) не существует). Пусть $d_{\{v_0,v_1,\ldots,v_{k-1}\},v_j}$ $(0 \leqslant j < k)$ хранит вес минимального гамильтонова пути, проходящего через все вершины $\{v_0, v_1, \ldots, v_{k-1}\}$, оканчивающегося в вершине v_j.

Тогда база выглядит как $d_{\{u\},u} = 0$ для всех u, а рекуррентная формула имеет вид

$$d_{\{v_0,v_1,\ldots,v_{k-1}\},v_j} = \min\{d_{\{v_0,v_1,\ldots,\widehat{v_j},\ldots,v_{k-1}\},v_0} + w_{v_j,v_0},$$
$$d_{\{v_0,v_1,\ldots,\widehat{v_j},\ldots,v_{k-1}\},v_1} + w_{v_j,v_1}, \ldots, d_{\{v_0,v_1,\ldots,\widehat{v_j},\ldots,v_{k-1}\},v_{j-1}} + w_{v_j,v_{j-1}},$$
$$d_{\{v_0,v_1,\ldots,\widehat{v_j},\ldots,v_{k-1}\},v_{j+1}} + w_{v_j,v_{j+1}}, \ldots, d_{\{v_0,v_1,\ldots,\widehat{v_j},\ldots,v_{k-1}\},v_{k-1}} + w_{v_j,v_{k-1}}\}$$

Перебор подмножеств будем реализовывать с помощью *масок*. Идея в том, что любое подмножество $\mathcal{S} \subseteq \mathcal{M} := \{0, 1, \ldots, n-1\}$ мощности n

задаётся битовым числом $b(\mathcal{S}) := \overline{b_{n-1} \dots b_1 b_0}$, где

$$b_i = \begin{cases} 1, & \text{если } i \in \mathcal{S}, \\ 0, & \text{иначе.} \end{cases}$$

```
for (unsigned int mask = 1; mask < (1 << n); ++mask)
{
    for (int i = 0; i < n; ++i)
    {
        if (mask & (1 << i)) // содержит ли i-й элемент
        {
            if (mask == (1 << i))
                d[mask][i] = 0;
            else
            {
                for (int j = 0; j < n; ++j)
                    if (j != i && mask & (1 << j))
                        d[mask][i] = min(d[mask][i],
                        d[mask ^ (1 << j)][j] + w[i][j]);
            }
        }
    }
}
```

Асимптотика этого алгоритма есть $O(2^n \cdot n^2)$, так что укладываться в одну секунду он будет (примерно) лишь на $n \leqslant 18$.

9 Кратчайшие пути

9.1 Алгоритм Дейкстры

Пусть имеем взвешенный граф (все веса положительные!). Есть некоторая вершина, от которой мы хотим найти кратчайшие пути до всех остальных вершин с минимальным суммарным весом.

В начальной вершине мы точно знаем ответ — 0. Мы смотрим все соседние стартовой вершины и говорим, что туда мы точно можем добраться за вес соединяющего их ребра (изначально во всех вершинах стоит $+\infty$). Теперь выбираем вершину с наименьшим посчитанным на этом этапе путём. Утверждается, что этот путь для неё действительно наименьший (не может уменьшиться при дальнейшем исполнении алгоритма). И из этой вершины обновляем все ответы, которые она может улуч-

шить. Можно также сохранять ребро, которое последний раз обновило путь к данной вершине (для построения дерева кратчайших путей).

Выбирать кратчайшую вершину можно полным перебором вершин. Тогда асимптотика составляет $O(n^2+m)$. Эта версия алгоритма выгодна, если $m \sim n^2$ (много рёбер и много обновлений весов).

Можно пользоваться структурой данных по типу множества или приоритетной очереди для выбора ребра с минимальным весом и обновления весов (вот из-за этого работает медленно, если много рёбер). Тогда асимптотика составит $O(n \log n + m \log n)$.

9.1.1 Реализация

Реализация без очереди за $O(n+m^2)$:

```
// Список смежности графа (храним также вес ребра в вершину)
vector<vector<pair<int, int>>> adj;
vector<int> dist(n, INF), marks(n, 0);

dist[start] = 0;
for (int i = 0; i < n; ++i)
{
    int minn = INF, min_v = -1;
    for (int j = 0; j < n; ++j)
    {
        if (!marks[j] && dist[j] < minn)
            minn = dist[j], min_v = j;
    }
    marks[min_v] = 1;
    for (auto u : adj[min_v])
    {
        if (dist[u.first] > dist[min_v] + u.second)
            dist[u.first] = dist[min_v] + u.second;
    }
}

```

Реализация с очередью за $O(m \log m)$:

```
vector<vector<pair<int, int>>> adj;
vector<int> dist(n, INF), marks(n, 0);

dist[start] = 0;
```

```
5
6   priority_queue<pair<int, int>> q;
7   for (int i = 0; i < n; ++i)
8       // очередь с приоритетом в начале хранит максимум,
9       // а нам нужен минимум, поэтому храним дистанцию
10      // со знаком «минус»
11      q.emplace(-dist[i], i);
12
13  while (!q.empty())
14  {
15      int v;
16      v = q.top().second;
17      q.pop();
18      if (!marks[v])
19      {
20          marks[v] = 1;
21          for (auto u : adj[v])
22          {
23              if (dist[u.first] > dist[v] + u.second)
24              {
25                  dist[u.first] = dist[v] + u.second;
26                  q.emplace(-dist[u.first], u.first);
27              }
28          }
29      }
30  }
```

9.2 Алгоритм Форда — Беллмана

Алгоритм позволяет находить кратчайшие пути в графах с отрицательными весами рёбер. Заметим, что мы работаем без отрицательных циклов (циклов с отрицательной суммой входящих в него рёбер). Ведь тогда можно бесконечно ходить по нему и уменьшать расстояние.

Лемма 1. Если существует ребро $y \to x$ с весом $w(y, x)$, то выполняется

$$\text{dist}_x \leqslant \text{dist}_y + w(y, x).$$

Лемма 2. Если в пути есть цикл, то можем его отбросить (т. к. он только увеличивает суммарный вес). Следовательно кратчайший путь содержит не более $n - 1$ ребра, т.к. в нём нет повторяющихся вершин.

Изначально в стартовой вершине вес 0, в остальных $+\infty$. Будем ходить по всем вершинам и обновлять расстояние по лемме 1. Утверждается,

что после i-го обхода все кратчайшие пути, содержащие i рёбер, будут определены. Тогда нам нужно совершить $n-1$ обход и мы получим все кратчайшие пути.

9.2.1 Реализация

```
// Список смежности графа (храним также вес ребра в вершину)
vector<vector<pair<int, int>>> adj;
vector<int> dist(n, INF);

dist[start] = 0;

for (int i = 0; i < n - 1; ++i)
{
    for (int v = 0; v < n; ++v)
    {
        for (auto u : adj[v])
            if(dist[u.first] > dist[v] + u.second)
                dist[u.first] = dist[v] + u.second;
    }
}
```

Может показаться, что асимптотика алгоритма больше $O(n^2 \cdot m)$, но на самом деле весь внутренний цикл (идём по v) — это $O(m)$. Таким образом, общая асимптотика $O(n^2 + n \cdot m)$.

Оптимизированная версия алгоритма Форда — Беллмана:

```
// Список смежности графа (храним также вес ребра в вершину)
vector<vector<pair<int, int>>> adj;
vector<int> dist(n, INF);
bool any = 1;
int cnt = 0;

dist[start] = 0;

while (any && cnt < n)
{
    any = 0, cnt++;
    for (int v = 0; v < n; ++v)
    {
        for (auto u : adj[v])
            if (dist[u.first] > dist[v] + u.second)
            {
```

```
                dist[u.first] = dist[v] + u.second;
                any = 1;
            }
    }
}
```

Теперь если мы ничего не обновили на каком-то этапе, то мы сразу выходим. Если мы что-то изменили на последней итерации, то у нас есть отрицательный цикл.

9.3 Алгоритм Флойда — Уоршелла

Здесь находим двумерный массив `dist[i][j]` размера $n \times n$, хранящий все расстояния между вершинами.

Лемма 3 (Неравенство треугольника)**.** $\text{dist}_{i,j} \leqslant \text{dist}_{i,k} + \text{dist}_{k,j}$.

Лемма 4. $\text{dist}_{i,i} = 0$.

Лемма 5. $\text{dist}_{u,v} \leqslant w(u,v)$.

9.3.1 Реализация

```
// Список смежности графа (храним также вес ребра в вершину)
vector<vector<int>> dist(n, vector<int>(n, INF));

for (int i = 0; i < m; ++i)
{
    cin >> u >> v >> w;
    // Выбираем наименьшее кратное ребро
    if (dits[u][v] > w)
        dist[u][v] = w;
}

for (int i = 0; i < n; ++i)
    dist[i][i] = 0;

for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (dist[i][j] > dist[i][k] + dist[k][j])
                dist[i][j] = dist[i][k] + dist[k][j];
```

Приведём 2 схемы доказательства корректности алгоритма:

1. **Индукция:** после k-го прохода имеем «настоящие» кратчайшие пути длины не более $k+1$, содержащие как промежуточные только вершины $\{0, 1, \ldots, k\}$. Таким образом, через n проходов мы получим все кратчайшие пути.

2. **От противного:** пусть есть ненайденные пути. Из них возьмём кратчайший (по количеству входящих в него рёбер) $i \to j$, а на нём возьмём вершину с максимальным номером m ($m > \min\{i, j\}$, так как иначе этот путь точно найден). Тогда не найден один из путей $i \to m$ или $m \to j$. Причём, оба этих пути короче $i \to j$. Противоречие, так как брали $i \to j$.

Асимптотика составляет $O(n^3)$, но это куб с маленькой константой, ведь мы не используем почти никаких дополнительных структур данных (мы даже сам граф не храним, мы храним только матрицу `dist`).

10 Битовый бой

10.1 Двоичная система счисления

Определение 1 (Системы счисления)**.**

$$\overline{a_{n-1} \ldots a_1 a_0}_x := \sum_{i=0}^{n-1} a_i \cdot x^i$$

В компьютере числа хранятся в двоичной системе счисления, причём хранение отрицательных чисел производится с помощью, так называемого, *обратного кода*. Первый бит отвечает за знак числа (для неотрицательных он равен 0, для отрицательных 1). Пусть имеем двоичное число $A = 1101\underbrace{0\ldots0}_{\text{28 нулей}}$. Как найти обратное к нему? Чтобы арифметика работала, нужно, чтобы такое число B при сложении с A давало 0. Этого можно добиться переполнением, то есть, нам нужно получить число $1\underbrace{0\ldots0}_{\text{32 нуля}}$, что в типе `int` является нулём (ведь он хранит только 32 младших бита). Так, нам подойдёт число $B = 0011\underbrace{0\ldots0}_{\text{28 нулей}}$.

10.2 Битовые операции

Определение 2.

1. *Конъюнкция* — побитовое «и» ($x \,\&\, y$);
2. *Дизъюнкция* — побитовое «или» ($x \mid y$);
3. *XOR* — сложение mod 2 ($x \wedge y$);
4. *Инверсия* — побитовое отрицание ($\sim x$);
5. *Побитовый сдвиг вправо* — умножение на 2^k (x << k);
6. *Побитовый сдвиг влево* — целочисленное деление на 2^k (x >> k).

$$\begin{array}{r} \& \;\; 110100 \\ \hline 100100 \end{array} \qquad \begin{array}{r} \mid \;\; 110100 \\ \hline 110111 \end{array} \qquad \begin{array}{r} \wedge \;\; 110100 \\ \hline 010011 \end{array} \qquad \begin{array}{r} \sim 110100 \\ \hline 001011 \end{array}$$

Задача 1. Определить значение i-го бита числа x

$\triangleright$ $x \,\&\, (1 << i)$

Задача 2. Инвертировать i-й бит числа x

$\triangleright$ $x \wedge (1 << i)$

10.3 Битовые маски

Заметим, что любое множество (из не более чем 32 элементов) можно задавать двоичной строкой (элементы кодируются 1 на соответствующих местах). Например, строка 0011001 задаёт множество $\{0, 3, 4\}$. Заметим, что это даёт нам возможность быстро использовать операции над множествами (им соответствую операции над двоичными числами). Пусть f — функция, которая по битовой маске строит соответствующее ей множество, A и B — некоторые множества. Некоторые примеры операций:

$$\begin{array}{l} A \cap B \leftarrow f^{-1}(A) \,\&\, f^{-1}(B), \\ A \cup B \leftarrow f^{-1}(A) \mid f^{-1}(B), \\ A \,\Delta\, B \leftarrow f^{-1}(A) \;\wedge\; f^{-1}(B), \\ \overline{A} \leftarrow \sim f^{-1}(A), \\ A \setminus B \leftarrow f^{-1}(A) \,\&\, (\sim f^{-1}(B)). \end{array}$$

10.4 Bitset-ы

Т. к. часто приходится работать с большими множествами (количество элементов в которых больше 32 и даже 64), нам не подходят для хранения обычные числовые типы вроде `int` и `long long`. Как раз для этого

существует `bitset<SIZE>` (`SIZE` — обязательно константа). Он предоставляет все операции над двоичными числами (единственное условие — они должны быть одинакового размера).

Помимо увеличенного размера двоичных чисел, `bitset`-ы нужны, чтобы уменьшать время работы операций, так как в нём биты хранятся не отдельно, а в составе байта (группами по 8 штук), поэтому операции с `bitset`-ом работают с очень маленькой константой (как раз в 8 раз меньше, чем при применении битовых операций к обычным числам).

11 Наименьший общий предок

11.1 Свойства DFS

Посчитаем для каждой вершины времена входа и выхода при обходе в глубину:

```
vector<vector<int>> adj;
vector<int> tin, tout;
int t = 0

void dfs(int u)
{
    tin[v] = t++;
    for (int v : adj[u])
        dfs(v);
    tout[u] = t; // можно и здесь увеличивать счётчик
}
```

Теорема 1.

1. Вершина u является предком $v \iff \text{tin}_v \in [\text{tin}_u; \text{tout}_u)$;
2. Два полуинтервала — $[\text{tin}_v; \text{tout}_v)$ и $[\text{tin}_u; \text{tout}_u)$ либо не пересекаются, либо один вложен в другой;
3. В массиве tin есть все числа из $[0; n)$, причём у каждой вершины свой номер;
4. Размер поддерева вершины v (включая саму v) равен $\text{tout}_v - \text{tin}_v$;

11.2 Основная задача и наивное решение

Задача 1. Дано корневое дерево. Требуется отвечать на запросы нахождения наименьшего общего предка вершин u_i и v_i, т. е. вершины w, кото-

рая лежит на пути корня до u_i, на пути от корня до v_i, и при этом самую глубокую (нижнюю) из всех таких.

За $O(n)$ наименьшего общего предка можно искать так:

```
// Проверить, является ли u предком v
bool a(int u, int v)
{
    return tin[u] <= tin[v] && tin[v] < tout[u];
}

int lca(int u, int v)
{
    while (!a(u, v))
        u = p[u];
    return u;
}
```

11.3 Двоичные подъёмы

Предпосчитаем для каждой вершины её 2^i-их предков и сохраним их в двумерном массиве up размера $n \times \lceil \log_2 n \rceil$: в $\text{up}_{v,d}$ будет храниться предок вершины v на расстоянии 2^d, а если его не существует, то корень.

Такой предпосчёт можно выполнить за $O(n \log n)$, используя тот факт, что предок на расстоянии 2^{d+1} — это предок на расстоянии 2^d предка на расстоянии 2^d.

```
vector<vector<int>> up;
// В переменной logn будем хранить ⌈log2 n⌉

void dfs(int u)
{
    for (int l = 1; l < logn; ++l)
        up[u][l] = up[up[u][l - 1]][l - 1];
    tin[u] = t++;
    for (int v : adj[u])
    {
        if (v != up[u][0])
        {
            up[v][0] = u;
            dfs(v);
        }
    }
```

```
17      tout[u] = t;
18  }
```

Пусть теперь поступил запрос нахождения наименьшего общего предка пары вершин (u, v):

1. Проверим, не является ли одна вершина предком другой — в таком случае она и является результатом;
2. Иначе, пользуясь массивом up, будем подниматься по предкам одной из них, пока не найдём самую высокую вершину, которая ещё не является предком другой. Следующая за ней будет искомым наименьшим общим предком.

Подробнее про второй пункт. Присвоим $i = \lceil \log_2 n \rceil$ и будем уменьшать эту переменую на единицу, пока $\mathrm{up}_{v,i}$ не перестанет быть предком u. Когда это произойдёт, подвинем указатель на 2^i-го предка v и продолжим дальше.

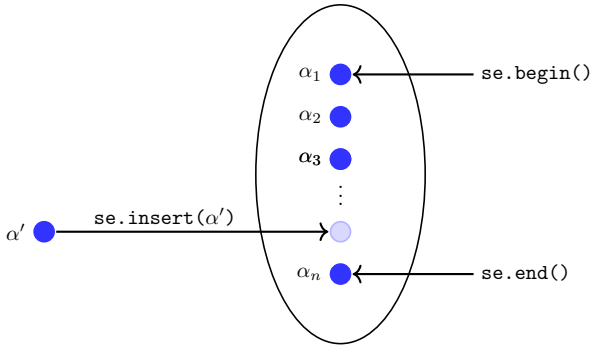
```
1   int lca(int v, int u)
2   {
3       if (a(v, u)) return v;
4       if (a(u, v)) return u;
5       for (int l = logn - 1; l >= 0; --l)
6           if (!a(up[v][l], u))
7               v = up[v][l];
8       return up[v][0];
9   }
```

Указатель $\mathrm{up}_{v,i}$ изначально является корнем дерева, а затем будет каждую итерацию спускаться на 2^i. Когда он станет потомком искомого общего предка, нам достаточно подняться всего лишь один раз, потому что два раза прыгнуть на расстояние 2^i — это то же самое, что один раз прыгнуть на 2^{i+1}, а мы могли это сделать на предыдущем шаге.

Предпосчёт занимает $O(n \log n)$, потому что таков размер массива up, и каждый его элемент вычисляется за константу. Ответ на произвольный запрос будет работать за $O(\log n)$, потому что фактически мы делаем один бинарный поиск.

12 STL

12.1 Множество

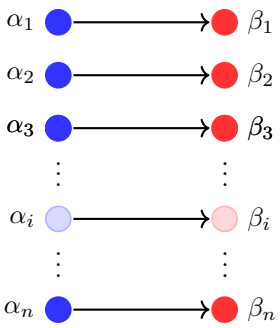


Множетсво инициализируется как `set<T> se`. Представляет собой стандартное математическое множетсво. Можно добавлять элементы с помощью `se.insert(`x`)`, удалять с помощью `se.erase(`x`)` или `se.erase(`p_x`)`, где p_x — указатель на элемент со значением x. Методы `se.begin()` и `se.end()` возвращают указатели на начальный и конечный элемент множества соответственно. Начальный элемент в множестве `int`-ов является минимальным в множестве. Метод `se.find(`x`)` возвращает указатель на элемент со значением x, то есть p_x. Если же такого элемента в множестве нет, то вернётся указатель на последний элемент — `se.end()`. В STL есть такая структура данных, как мультимножество, `multiset<T> mse`. Оно отличается от обычного множества тем, что может хранить неуникальные элементы. Теперь удаление элемента по значению влечёт удаление всех его вхождений.

Все операции работают за $O(\alpha \cdot \log n)$, где n — размер множества, а α — оценка сравнения ключей. Например, для целых чисел $\alpha = O(1)$, а для строк $\alpha = O(|s|)$.

12.2 Карта/словарь

Карта представляет собой набор пар (α_i, β_i), в которых по умолчанию $\beta_i = 0$. Карта инициализируется как `map<T1, T2> ma`. При этом, к элементам можно обращаться следующим образом: `ma[`α_i`]` $= \beta_i$. Пары внутри `map`-а (как и элементы внутри `set`-а) сортируются по возрастанию. А пары сортируются по первому элементу, поэтому `ma.begin()` указывает на пару с наименьшим первым элементом.



Все операции в `map`-е работают за $O(\log n)$, даже получение элемента. Поэтому стоит минимизировать их количество. Контейнер `map` в полную силу раскрывается, если в качестве ключей использовать строки. Тогда мы строкам можем очень быстро сопоставлять числа. Хранить в массиве это было бы невозможно, так как строки могут состоять из очень большого количества символов; в `map`-е же такой проблемы не возникает.

12.3 Очередь с приоритетом

STL предоставляет также контейнер `priority_queue`, который, помимо функционала обычной очереди, поддерживает определённый порядок элементов: в голове очереди (по `q.front()`) находится наибольший элемент. Инициализируется так: `priority_queue<T> q`, поддерживает методы `front()` за $O(1)$ и `push_back()`, `pop_back()` за $O(\log n)$.

13 Мосты, точки сочленения, компоненты сильной связности

13.1 Мосты

Определение 1. *Мостом* называется ребро, при удалении которого число компонент связности неориентированного графа увеличивается.

Далее мы рассматриваем граф как одну компоненту связности и ищем мосты внутри неё.

Запустим обход в глубину из произвольной вершины. Введём новые виды рёбер:

1. *Прямые* — те, по которым были переходы;
2. *Обратные* — все остальные.

Заметим, что никакое обратное ребро (u, v) не может являться мостом: если его удалить, то всё равно будет существовать какой-то путь от u до v, потому что подграф из прямых рёбер является связным деревом.

Значит, остаётся проверить только все прямые рёбра. Заметим, что обратные рёбра могут вести только «вверх» — к какому-то предку в дереве обхода графа, но не в другие «ветки», ведь иначе наш обход увидел бы это ребро раньше, и оно было бы прямым.

Тогда, чтобы определить, является ли прямое ребро $v \to u$ мостом, мы можем воспользоваться следующим критерием: глубина h_v вершины v меньше, чем минимальная глубина всех вершин, соединённых обратным ребром с какой-либо вершиной из поддерева u.

Для ясности, обозначим эту величину как d_u, её можно считать во время обхода по следующей формуле:

$$d_v = \min \begin{cases} h_v, \\ d_u, & \text{ребро } v \to u \text{ прямое}, \\ h_u, & \text{ребро } v \to u \text{ обратное}. \end{cases}$$

Если это условие ($h_v < d_u$) не выполняется, то существует какой-то путь из u в какого-то предка v или саму v, не использующий ребро (v, u), а в противном случае — наоборот.

```
vector<int> used, h, d;

void dfs(int v, int p = -1)
{
    used[v] = 1;
    d[v] = h[v] = (p == -1 ? 0 : h[p] + 1);
    for (int u : adj[v])
    {
        if (u != p)
        {
            if (used[u]) // если ребро обратное
                d[v] = min(d[v], h[u]);
            else // если ребро прямое
            {
                dfs(u, v);
                d[v] = min(d[v], d[u]);
                if (h[v] < d[u])
                {
                    // Если нельзя другим путём добраться
                    // в v или выше, то ребро (v,u) - мост
```

```
                }
            }
        }
    }
}
```

13.2 Точки сочленения

Определение 2. *Точкой сочленения* называется вершина, при удалении которой связный неориентированный граф становится несвязным.

Задача поиска точек сочленения не сильно отличается от задачи поиска мостов.

Вершина v является точкой сочленения, когда из какого-то её сына u нельзя дойти до её предка, не используя ребро (v, u). Для конкретного прямого ребра $v \to u$ этот факт можно проверить так: $h_v \leqslant d_u$ (теперь неравенство нестрогое, т. к. если из вершины можно добраться только неё самой, то она всё равно будет точкой сочленения).

Используя этот факт, можно оставить алгоритм практически прежним — нужно проверить этот критерий для всех прямых рёбер $v \to u$.

```
void dfs(int v, int p = -1)
{
    used[v] = 1;
    d[v] = h[v] = (p == -1 ? 0 : h[p] + 1);
    int children = 0;
    for (int u : adj[v])
    {
        if (u != p)
        {
            if (used[u])
                d[v] = min(d[v], h[u]);
            else
            {
                dfs(u, v);
                d[v] = min(d[v], d[u]);
                if (h[v] <= d[u] && p == -1)
                {
                    // v - точка сочленения
                    // (это условие может выполняться
                    // много раз для разных детей)
                }
```

```
22                  ++children;
23              }
24          }
25      }
26      // Корень смотрим отдельно
27      if (p == -1 && children > 1)
28      {
29          // v - корень и точка сочленения
30      }
31  }
```

Единственный крайний случай — это корень, т. к. в него мы войдём раньше других вершин. Поправить просто — достаточно посмотреть, было ли у него более одной ветви в обходе (если корень удалить, то его поддеревья станут несвязными между собой).

13.3 Топологическая сортировка

Задача 1. Дан ориентированный ациклический граф. Требуется найти такой порядок вершин, в котором все рёбра графа вели из более ранней вершины в более позднюю.

Во-первых, заметим, что граф с циклом топологически отсортировать не получится — как ни располагай цикл в массиве, всё время идти вправо по рёбрам цикла не получится. Во-вторых, верно обратное — если цикла нет, то его обязательно можно топологически отсортировать.

Лемма 1. В ориентированном графе либо есть цикл, либо вершина без выходящих рёбер.

Доказательство. Выберем произвольную вершину и начнём с неё ол переходить по рёбрам графа. Либо этот процесс когда-нибудь закончится (если когда-нибудь попадём в вершину без выходящих рёбер), либо будет продолжаться бесконечно (но тогда, в силу конечности числа вершин в графе, попадём в какую-то вершину дважды). ■

Заметим, что вершину, из которой не ведёт ни одно ребро, можно всегда поставить последней, а такая вершина в ациклическом графе всегда есть. Из этого сразу следует конструктивное доказательство: будем итеративно класть в массив вершину, из которой ничего не ведёт, и убирать её из графа. Новых циклов при этом, очевидно, не появится, так что в новом графе опять найдётся вершина без выходящих рёбер. После этого процесса массив надо будет развернуть.

Этот алгоритм проще реализовать, обратив внимание на времена выхода вершин в поиске в глубину. Вершина, из которой мы выйдем первой — та, у которой нет новых исходящих рёбер. Дальше мы будем выходить только из тех вершин, которые если и имеют исходящие рёбра, то только в те вершины, из которых мы уже вышли.

Следовательно, достаточно просто выписать вершины в порядке выхода из обхода в глубину, а затем полученный список развернуть, и мы получим одну из корректных топологических сортировок.

```
vector<vector<int>> adj;
vector<int> used, t;

void dfs(int u)
{
    used[u] = 1;
    for (int v : adj[u])
        if (!used[v])
            dfs(v);
    t.push_back(u);
}

void top_sort()
{
    for (int u = 0; u < n; ++u)
        if (!used[u])
            dfs(u);
    reverse(t.begin(), t.end());
}
```

Топологическую сортировку можно использовать для проверки достижимости, сравнивая номера вершин в получившемся массиве. Факт того, что вершина a идёт позже вершины b, говорит о том, что из a недостижима b — однако a может быть как достижима, так и недостижима из b.

13.4 Компоненты сильной связности

Мы научились топологически сортировать ациклические графы. Но в циклических графах тоже иногда требуется найти какую-то структуру, для чего нам нужно ввести следующее понятие.

Определение 3. Две вершины ориентированного графа *сильно связны*, если существует путь из одной в другую, и наоборот. Иными словами,

они обе лежат в каком-то цикле.

Утверждение 1. Отношение сильной связности является отношением эквивалентности.

Доказательство. Рефлексивность и симметричность очевидны из определения, проверим транзитивность. Пусть a и b сильно связны, и b и c сильно связны. Тогда можно дойти из a в b, а из bдойти в c, и наоборот. Так что вершины a и c тоже сильно связны. ■

Определение 4. Классы этой эквивалентности называются *компонентами сильной связности.*

Самый простой пример сильно-связной компоненты — это цикл. Но это может быть и полный граф, или сложное пересечение нескольких циклов.

Часто рассматривают граф, составленный из самих компонент сильной связности, а не индивидуальных вершин. Очевидно, такой граф уже будет ациклическим, с ним проще работать. Задачу о сжатии каждой компоненты сильной связности в одну вершину называют *конденсацией* графа, и её решение мы сейчас опишем.

13.5 Конденсация графа

Если мы уже знаем, какие вершины лежат в каждой компоненте сильной связности, то построить граф конденсации несложно: нужно провести некоторые манипуляции со списками смежности, заменив для всех рёбер номера вершин номерами их компонент, а затем объединив списки смежности для всех вершин каждой компоненты. Поэтому сразу сведём исходную задачу к нахождении самих компонент.

Лемма 2. Запустим обход в глубину. Пусть A и B — две различные компоненты сильной связности, и пусть в графе конденсации между ними есть ребро $A \to B$. Тогда $\max\limits_{a \in A} \mathrm{tout}_a > \max\limits_{b \in B} \mathrm{tout}_b$.

Доказательство. Рассмотрим два случая, в зависимости от того, в какую из компонент обход зайдёт первым:

1. Пусть первой была достигнута компонента A, т. е. в какой-то момент времени DFS заходит в некоторую вершину v компоненты A, и при этом все остальные вершины компонент A и B ещё не посещены. Но т. к. по условию в графе конденсаций есть ребро $A \to B$, то из вершины v будет достижима не только вся компонента A, но и вся компонента B. Это означает, что при запуске из вершины v обход в глубину пройдёт по всем вершинам компонент A и B, а значит, они

станут потомками по отношению к v в дереве обхода, и для любой вершины $u \in A \sqcup B$, $u \neq v$ будет выполнено $\text{tout}_v > \text{tout}_u$, что и требовалось.

2. Второй случай проще: из B по условию нельзя дойти до A, а значит, если первой была достигнута B, то DFS выйдет из всех её вершин ещё до того, как войти в A.

■

Из этого факта следует первая часть решения. Отсортируем вершины по убыванию времени выхода (т. е. как бы сделаем топологическую сортировку, но на циклическом графе). Рассмотрим компоненту сильной связности первой вершины в сортировке. В эту компоненту точно не входят никакие рёбра из других компонент — иначе нарушилось бы условие леммы, ведь у первой вершины tout максимальный. Поэтому, если развернуть все рёбра в графе, то из этой вершины будет достижима своя компонента сильной связности C, и больше ничего — если в исходном графе не было рёбер из других компонент, то в развёрнутом графе не будет рёбер в другие компоненты.

После того, как мы сделали это с первой вершины, мы можем пойти по топологически отсортированному списку дальше и то же самое с вершинами, для которых компоненту связности мы ещё не отметили.

```
vector<vector<int>> adj, t;
vector<int> order, used, comp;
int cnt_comps = 0;

// Топологическая сортировка
void dfs1(int u)
{
    used[u] = 1;
    for (int v : adj[u])
        if (!used[v])
            dfs1(v);
    order.push_back(u);
}

void dfs2(int u)
{
    comp[u] = cnt_comps;
    for (int v : t[u])
        if (!comp[v])
            dfs2(v);
}
```

```
22
23  // ...
24  // В содержательной части main:
25
26  // Разворачиваем граф
27  for (int u = 0; u < n; ++u)
28      for (int v : adj[u])
29          t[v].push_back(u);
30
31  // Запускаем топологическую сортировку
32  for (int i = 0; i < n; ++i)
33      if (!used[i]) dfs1(i);
34
35  // Выделяем компоненты
36  reverse(order.begin(), order.end());
37  for (int u : order)
38      if (!comp[u])
39          dfs2(u), ++cnt_comps;
```

14 Бинарный и тернарный поиски

14.1 Основная идея

Предположим, есть массив $\{a_0, a_1, \ldots, a_{n-1}\}$ и некоторое монотонное свойство P в том смысле, что найдётся такой $i_0 \in [0; n)$, что $P(a_i) = P(a_j)$ для любых $i, j < i_0$ и для любых $i, j \geqslant i_0$, но $P(a_i) \neq P(a_j)$ для любых i, j, т. ч. $i < i_0 \leqslant j$.

Наша задача — найти этот индекс i_0. Мы будем делать это следующим образом. Зафиксируем начало и конец полуинтервала поиска: $l := 0$, $r := n$. Если $P\left(a_{\lfloor (l+r)/2 \rfloor}\right) = P(a_l)$, то индекс перехода i_0 находится где-то на второй половине полуинтервала; иначе — на первой. Обновляем значения l и r в соответствии с новым интервалом поиска. Алгоритм продолжает работу до тех пор, пока $l < r + 1$ (т. е. множество $[l; r)$ непусто).

Каждый раз интервал поиска уменьшается в два раза, так что совершаем $O(\log n)$ действий, а точнее — либо $\lceil \log_2 n \rceil$, либо $\lfloor \log_2 n \rfloor$. Наивное решение (пройти по всем элементам массива) имеет асимптотику $O(n)$.

14.2 Реализация

Задача 1. Найти в отсортированном массиве $\{a_0, a_1, \ldots, a_{n-1}\}$ элемент, равный x, или определить, что его там нет.

Положим $P(a_i) = \begin{cases} 1, & \text{если } a_i \geqslant x, \\ 0, & \text{иначе} \end{cases}$. Находим граничный элемент следующим образом:

```
int l = 0, r = n;
while (l < r + 1)
{
    int m = (l + r) / 2;
    if (a[m] > x)
        r = m;
    else
        l = m;
}
```

14.3 Вещественный бинарный поиск

Пусть теперь нам дана некоторая функция $f : \mathbb{R} \to \mathbb{R}$. Известно, что она непрерывна на отрезке $[a; b]$ и $f(a) \cdot f(b) < 0$. Теорема из математического анализа говорит, что у этой функции есть корень на данном отрезке. Искать его можно так же бинарным поиском, только теперь будем выполняться, пока $r - l > \varepsilon$, где ε выбирается исходя из необходимой в задаче погрешности.

```
long double l = a, r = b;
long double eps = 1e-6; // ... например
while (r - l > eps)
{
    long double m = (l + r) / 2;
    if (f(m) >= 0)
        r = m;
    else
        l = m;
}
```

А можно делать фиксированное количество итераций, при этом чтобы получить k правильных цифр, необходимо выполнить не менее $k \cdot \log_2 10$ операций.

14.4 Бинарный поиск по ответу

Задача 2. На прямой расположены n стойл (даны их координаты на прямой), в которые необходимо расставить k коров так, чтобы минимальное

расстояние между коровами было как можно больше. Гарантируется, что $1 < k < n$.

Если решать задачу в лоб, то вообще неясно, что делать. Вместо этого решим более простую задачу: предположим, что мы знаем это расстояние x, ближе которого коров ставить нельзя. Тогда сможем ли мы расставить всех коров в соответствие с условием? Заметим, что это свойство монотонно — для каких-то маленьких x коров точно можно расставить, а начиная с каких-то больших — уже нельзя, поэтому эту границу можно искать бинарным поиском.

```
bool check(int x)
{
    int cows = 1;
    int last_cow = a[0];
    for (int c : a)
    {
        if (c - last_cow >= x)
        {
            ++cows;
            last_cow = c;
        }
    }
    return (cows >= k);
}

int bin_search()
{
    // Предполагаем, что координаты
    // в массиве a уже отсортированы
    int l = 0, r = a.back() - a[0] + 1;
    while (r - l > 1)
    {
        if (check(m))
            l = m;
        else
            r = m;
    }
    return l;
}
```

Задача 3. Есть два принтера. Один печатает лист раз в x минут, другой — раз в y минут. За сколько минут они напечатают n листов?

Конечно, эту задачу можно решить формулой за $O(1)$. Однако если такую формулу придумать не получается, сведём задачу к обратной. Подумаем, как по числу минут t понять, сколько листов напечатается за это время? Очень легко:

$$\lfloor \frac{t}{x} \rfloor + \lfloor \frac{t}{y} \rfloor.$$

Ясно, что за 0 минут n листов наспечатать нельзя, а за $x \cdot n$ минут один только первый принтер успеет напечатать n листов. Поэтому 0 и $x \cdot n$ — это подходящие начальные границы для бинарного поиска.

14.5 Тернарный поиск

Пусть дана функция $f(x)$ имеем ровно один *экстремум* — локальный максимум или локальный минимум на отрезке $[a; b]$. Требуется найти точку этого экстремума.

Возьмём любые две точки m_1 и m_2 на этом отрезке: $l < m_1 < m_2 < r$. Посчитаем значения функции $f(m_1)$ и $f(m_2)$. Дальше у нас получается три варианта:

1. Если окажется, что $f(m_1) < f(m_2)$, то искомый максимум может находиться только в правой части, т. е. на отрезке $[m_1; r]$;
2. Если, наоборот, $f(m_1) > f(m_2)$, то искомый максимум может находиться только в левой части, т. е. на отрезке $[l; m_2]$;
3. А если $f(m_1) = f(m_2)$, то максимум находится на отрезке $[m_1; m_2]$, однако этот случай в целях упрощения кода можно отнести к любому из двух предыдущих.

Выполнение алгоритма завершается, когда отрезок поиска станет достаточно маленьким. Осталось заметить, что мы не накладывали никаких ограничений на выбор точек m_1 и m_2. От этого способа будет зависеть скорость сходимости (и возникающая погрешность). Можно выбирать точки так, чтобы отрезок $[l; r]$ делился ими на 3 равные части:

$$m_1 = l + \frac{r-l}{3}, \quad m_2 = r - \frac{r-l}{3}.$$

При этом каждый раз отрезок будет уменьшаться в 3/2 раза. Однако, можно выбирать m_1 и m_2 очень близко друг к другу (и к середине отрезка), тогда каждый раз будем уменьшать отрезок почти в 2 раза.

В случае целочисленного аргумента, отрезок $[l; r]$ становится дискретным, однако в силу отсутствия ограничений на выбор точек m_1 и m_2 корректность алгоритма не нарушается. Критерий остановки теперь — $r - l < 3$, ведь в таком случае уже невозможно выбрать точки m_1 и m_2 удовлетворяли необходимому нам условию.

15 Строки: p, z-функции, бор, хеши

15.1 Префикс-функция

Определение 1. *Префикс-функцией* от строки s называется массив p, где p_i равно длине самого большого префикса строки $[s_0 \dots s_i]$, который также является и её суффиксом (не считая весь i-й префикс).

Алгоритм Кнута — Морриса — Пратта. Научимся искать подстроку s в строке t. Пока поверим, что мы умеем считать префикс-функцию за линейное от размер строки время, и научимся с помощью неё искать подстроку в строке. Соединим подстроки s и t каким-нибудь символом, который не встречается ни там, ни там — пусть это символ $. Ясно, что все места, где значения перфикс-функции равны $|s|$, — это концы вхождений s в текст t.

Чтобы быстро считать префикс-функцию, докажем следующий факт.

Лемма 1. $p_{i+1} - p_i \leqslant 1$.

Доказательство. Если есть префикс, равный суффиксу строки $[s_0 \dots s_{i+1}]$, длины p_{i+1}, то, отбросив последний символ, можно получить правильный суффикс для строки $[s_0 \dots s_i]$, длина которого будет ровно на единицу меньше. ■

Попытаемся решить задачу с помощью динамики: найдём формулу для p_i через предыдущие значения. Заметим, что $p_{i+1} = p_i + 1$ в том и только том случае, когда $s_{p_i} = s_{i+1}$. В этом случае мы можем просто обновить p_{i+1} и пойти дальше. Но что происходит, когда $s_{p_i} \neq s_{i+1}$? Тогда проверяем равенство $p_{i+1} = p_{p_i} + 1$. Если оно неверно, то проверяем $p_{i+1} = p_{p_{p_i}} + 1$ и т. д. Делаем так, пока индекс не станет нулевым.

```
vector<int> prefix_function(string s)
{
    int n = (int)s.size();
    vector<int> p(n, 0);
    for (int i = 1; i < n; ++i)
    {
        // Префикс-функция точно не больше p_{i-1} + 1
        int j = p[i - 1] + 1;
        // Уменьшаем j, пока новый символ не подойдёт
        while (s[i] != s[j] && cur > 0)
            j = p[j - 1];
        // Здесь либо s_i = s_j, либо j = 0
```

```
13          if (s[i] = s[j])
14              p[i] = j + 1;
15      }
16      return p;
17  }
```

В худшем случае `while` может работать за $O(n)$, но в среднем каждый `while` работает за $O(1)$.

Префикс-функция каждый шаг возрастает максимум на 1 и после каждой итерации `while` хотя бы на 1. Значит, суммарно операций будет не более $O(n)$.

15.2 z-функция

Определение 2. *z-функция* от строки s определяется как массив z такой, что z_i равно длине максимальной подстроки, начинающейся с i-ой позиции, которая равна префиксу s.

z-функцию можно использовать вместо префикс-функции в алгоритме Кнута — Морриса — Пратта — только теперь нужные позиции будут начинаться с $|s|$, а не заканчиваться.

Чтобы быстро считать z-функцию, будем идти слева направо и хранить *z-блок* — саму правую подстроку, равную префиксу, которую мы успели обнаружить. Будем обозначать его границы как l и r включительно.

Пусть мы сейчас хотим найти z_i, а все предыдущие уже нашли. Новый i-й символможет лежать либо правее z-блока, либо внутри него:

1. Если правее, то мы просто наивно перебором найдём z_i (максимальный отрезок, начинающийся с s_i и равный префиксу), и объявим его новым z-блоком;
2. Если i-й элемент лежит внутри z-блока, то мы можем посмотреть на значение z_{i-l} и использовать его, чтобы инициализировать z_i чем-то, возможно, отличным от нуля. Если z_{i-l} левее правой границы z-блока, то $z_i = z_{i-l}$ — больше z_i быть не может. Если он упирается в границу, то «обрежем» его до неё и будем увеличивать на 1.

```
1   vector<int> z_function(string s)
2   {
3       int n = (int)s.size();
4       vector<int> z(n, 0);
5       int l = 0, r = 0;
6       for (int i = 1; i < n; ++i)
```

```
    {
        // Если мы уже видели этот символ
        if (i <= r)
        {
            // То мы можем попробовать его инициализировать
            // z_{i-l}, но не дальше правой границы
            z[i] = min(r - i + 1, z[i - l]);
        }
        // Дальше каждое успешное увеличение z_i
        // сдвинет z-блок на 1
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
        {
            r = i + z[i] - 1;
            l = i;
        }
    }
    return z;
}
```

В алгоритме мы делаем столько же действий, сколько раз сдвигается правая граница z-блока, т. е. $O(n)$.

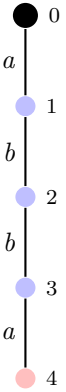
15.3 Бор

Задача 1. Пусть нам дан словарь из n слов s_i, и нам нужно отвечать на q запросов вида s: нужно понять, если в s в словаре. Можно решать бинарным поиском, тогда каждый запрос обрабатывается за $O(|s| \log n)$, с помощью бора мы научимся решать задачу за $O(|s|)$.

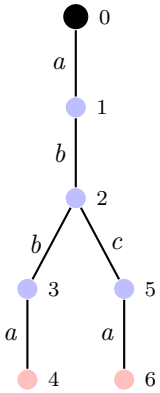
15.3.1 Основная идея

Изначально бор представляет из себя один корень. Мы хотим добавить к бору первое слово. Для этого мы добавляем его по буквам в граф (каждая буква является вершиной, рёбрами соединены вершины, если они являются соседними в каком-либо слове). А потом каждый раз, когда мы хотим добавить слово, мы идём по общему префиксу, пока можем, а затем создаём ответвление. Можно усложнять бор, например, в каждой вершине хранить количество терминальных вершин после неё (терминальная вершина — та, в которой заканчивается слово). Тогда мы можем для любого префикса узнать количество слов с таким префиксом.

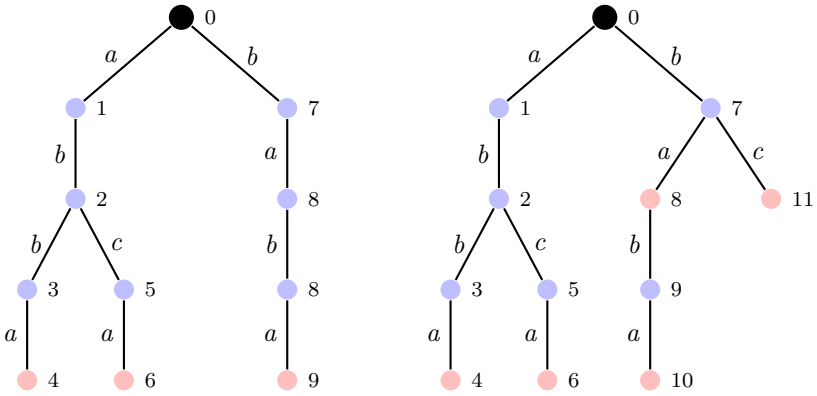
Рассмотрим пример построения бора на словаре $\{abba, abca, baba, bc, ba\}$. Будем поочерёдно добавлять слова к дереву. Изначально у нас есть только корень, к нему дописываем слово a — b — b — a.



Получили такой «бамбук». Теперь в дереве мы имеем перфиксы $\varnothing$ (корень), a, a — b, a — b — b и a — b — b — a. У следующего слова с текущим деревом есть общий префикс a — b, поэтому мы будем не создавать новую ветку от корня, а идти по старой до конца общего префикса. Тогда наш бор приобретает следующий вид:



Следующее слово не имеет общих префиксов с уже добавленными (на самом деле имеет, просто пустой), поэтому его подвешиваем к корню:



Аналогично добавляем в бор остальные слова. Заметим, что терминальные вершины (помечены красным на иллюстрациях) не обязательно должны быть листьями деревьев. Теперь научимся по данному префиксу находить количество слов с таким префиксом. Будем определять префикс по его конечной букве (конечной вершине бора). Понятно, что для листовых вершин (которые обязательно являются терминальными) ответ — 1. А для каждой следующей действует следующее правило: искомое число равно сумме входящих значений входящих в неё вершин. Также нужно прибавить 1, если данная вершина является терминальной.

Следует также помнить, что бор можно строить не только на строках, а на двоичных представлениях чисел. В такой структуре вставка и поиск работают за $O(|n_2|)$, где n_2 — двоичная строка, представляющая число n.

15.3.2 Реализация

```
// нам нужны ссылки на все буквы
vector<vector<int>> link(1, vector<int>(26, -1));
// для каждой вершины отмечаем, является ли она терминальной
vector<int> marks(1, 0);

// функция для вставки слова в бор
// передаём не саму строку, а ссылку на неё, это работает за
// O(1), а не O(|s|)
void insert(string& s)
{
    int v = 0;
    for (int i = 0; i < int(s.size()); ++i)
    {
```

```
14          int to = s[i] - 'a';
15          if (link[v][to] == -1)
16          {
17              // Создаём новую вершину
18              link.emplace_back(26, -1);
19              link[v][to] = int(marks.size()) - 1;
20              marks.push_back(0);
21          }
22          v = link[v][to]; // переходим к новой вершине
23      }
24      marks[v]++;
25  }
26
27  bool find(string& s)
28  {
29      int v = 0;
30      for (int i = 0; i < int(s.size()); ++i)
31      {
32          int to = s[i] - 'a';
33          if (link[v][to])
34              return false;
35      }
36
37      return marks[v];
38  }
```

15.4 Хеши

Определение 3 (*Прямой полиномиальный хеш*). $h(s) := s_0 + s_1 k + s_2 k^2 + \ldots + s_n k^n \pmod p$, где k — произвольное число, большее размера алфавита, а p — достаточно большой модуль.

Его можно подсчитать за линейное время, поддерживая переменную, равную k в нужной степени:

```
1  long long h = 0, m = 1;
2
3  for (char c : s)
4  {
5      int x = (int)(c - 'a' + 1);
6      h = (h + m * x) % mod;
7      m = (m * k) % mod;
8  }
```

Используя тот факт, что хеш — это значение многочлена, можно быстро пересчитывать хеш от результата выполнения многих строковых операций. Например, так можно посчитать хеш от конкатенации строк a и b: $h(ab) = h(a) + k^{|a|} \cdot h(b)$. Удалить префикс строки можно так: $h(b) = \frac{h(ab)-h(a)}{k^{|a|}}$, суффикс — так: $h(a) = h(ab) - k^{|a|} \cdot h(b)$.

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

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

```
vector<int> h;
h[0] = 0;

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

Теперь с помощью этих префиксных хешей мы можем определить функцию, которая будет считать хеш на любом подотрезке: $h([s_l \dots, s_r]) = = \frac{h_r-h_l}{k^l}$. Деление по модулю возможно делать только при взаимно простых k и модуля. В любом случае, можно этого избежать. Для нашей задачи неважно получать именно полиномиальный хеш — достаточно, чтобы наша функция возвращала одинаковый многочлен от одинаковых подстрок. Вместо приведения к нулевой степени приведём многочлен к какой-нибудь достаточно большой, например, к n-й: $h([s_l \dots s_r]) = k^{n-l}(h_r - h_l)$. Так проще — теперь нужно домножать, а не делить:

```
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 $O(1)$.

16 Математика

16.1 Бинарное возведение в степень

Алгоритм бинарного возведения в степень основан на следующих очевидных свойствах:

$$x^{2n} = (x^2)^n, \quad x^{2n+1} = x \cdot (x^2)^n.$$

```
int bin_pow(int x, int n)
{
    if (n == 0) return 1;
    if (n % 2 == 0) return bin_pow(x * x, n / 2); // x^{2n} = (x^2)^n
    return x * bin_pow(x * x, n / 2); // x^{2n+1} = x * (x^2)^n
}
```

Таким образом каждый раз мы уменьшаем степень, в которую нужно возвести число, в 2 раза, значит функция отработает за $O(\log n)$.

Чтобы не возникло переполнения, следует каждую операцию брать по модулю p:

```
int bin_pow_modp(int x, int n)
{
    if (n == 0) return 1;
    if (n % 2 == 0) return bin_pow_modp((x * x) % p, n / 2);
    return (x * bin_pow_modp((x * x) % p, n / 2)) % p;
}
```

16.2 Алгоритм Евклида

Обозначим НОД(a, b) — наибольший общий делитель чисел a, b, а через НОК(a, b) — наименьшее общее кратное a и b. Алгоритм Евклида позволяет за $O(\log \min\{a, b\})$ находить НОД(a, b), основываясь на свойстве НОД(a, b) = НОД$(b, a \bmod b)$. Функция `gcd` принимает два аргумента: a и b, причём поддерживается $a \geqslant b$.

```
int gcd(int a, int b)
{
    if (a < b) swap(a, b);
    if (b == 0) return a;
    return gcd(b, a % b);
}
```

Наименьшее общее кратное можно найти с помощью соотношения

$$\text{НОД}(a, b) \cdot \text{НОК}(a, b) = a \cdot b.$$

Если обычный алгоритм Евклида возвращает НОД(a, b), то расширенный позволяет найти коэффициенты x и y такие, что $ax + by =$ НОД(a, b).

Для этого нам нужно понять, как эти числа меняются от пары (a, b) к паре $(b, a \bmod b)$. Пусть (x, y) — искомые коэффициенты для пары (a, b), (x_1, y_1) — для пары $(a \bmod b, b)$. Тогда

$$\begin{cases} \text{НОД}(a \bmod b, b) = b \cdot x_1 + (a \bmod b) \cdot y_1, \\ \text{НОД}(a, b) = x \cdot a + y \cdot b. \end{cases}$$

Откуда

$$\text{НОД}(a, b) = b \cdot x_1 + \left(a - \left\lfloor \frac{a}{b} \right\rfloor b\right) \cdot y_1 = y_1 \cdot a + \left(x_1 - y_1 \cdot \left\lfloor \frac{a}{b} \right\rfloor\right) \cdot b$$

```
tuple<int, int, int> ext_gcd(int a, int b)
{
    if (b == 0) return {a, 1, 0};
    int d, x1, y1;
    tie(d, x1, y1) = ext_gcd(b, a % b);
    return {d, y1, x1 - y1 * (a / b)};
}
```

С помощью расширенного алгоритма можно искать решения диофантовых уравнений, ведь все решения уравнения

$$a \cdot x + b \cdot y = \text{НОД}(a, b)$$

представляются в виде

$$\begin{aligned} x &= x_1 + \frac{a}{\text{НОД}(a, b)} \cdot k \\ y &= y_1 - \frac{b}{\text{НОД}(a, b)} \cdot k \end{aligned}, \quad k \in \mathbb{Z}.$$

А коэффициенты x_1, y_1 и $\text{НОД}(a, b)$ мы знаем из расширенного алгоритма Евклида.

16.3 Арифметика по модулю

Пусть m — натуральное число. Произвольное целое число можно поделить на m с остатком, который принимает значения $0, 1, \ldots, m-1$. Разобьём множество целых чисел на m классов, каждый из которых содержит все целые числа, дающие один и тот же остаток при делении на m. Это будут так называемые *классы вычетов по модулю* m.

Определение 1. *Обратным элементом* к a по модулю m называется такой элемент b, что $a \cdot b \equiv 1 \pmod{m}$.

Ясно, что обратных элементов к данному a может быть (бесконечно) много, однако все они попадают в один класс вычетов, который и обозначается через a^{-1}. Если m простое, то находить обратный элемент помогает

Теорема 1 (Малая теорема Ферма)**.** Если p простое и a целое, то

$$a^{p-1} \equiv 1 \pmod{p}.$$

Действительно, разделив на a обе части, получим $a^{p-2} \equiv a^{-1} \pmod{p}$. А быстро возводить в степень мы уже умеем.

Если же m составное, то обратный к a по модулю m существует тогда и только тогда, когда НОД$(a, m) = 1$. В этом случае его можно найти, решив диофантово уравнение

$$ax + my = 1,$$

например, с помощью расширенного алгоритма Евклида. Т. к. $my \equiv 0 \pmod{m}$, то $ax \equiv 1 \pmod{m}$, т. е. $x \equiv a^{-1} \pmod{m}$ по определению.

16.4 Биномиальные коэффициенты. Предпосчёт факториалов

Для быстрого вычисления биномиальных коэффициентов предлагается предпосчитать массив факториалов (по модулю p) следующим образом:

```
vector<int> fact(n, 1);

for (int i = 1; i < n; ++i)
    fact[i] = (fact[i - 1] * i) % p;
```

Также нам понадобится массив обратных факториалов:

```
vector<int> inv_fact(n, 1);

inv_fact[n - 1] = bin_pow(fact[n - 1], p - 2);

for (int i = n - 2; i >= 0; --i)
    inv_fact[i] = inv_fact[i + 1] * (i + 1);
```

17 Дерево Фенвика, разреженная таблица