Алгоритмы и структуры данных—2
Коллоквиум

Винер Даниил @danya_vin

Версия от 18 октября 2024 г.

Содержание

1 Хэш-функция. Полиномиальное хэширование

1.1 Хэш-функция

Определение. Хэш-функцией называется функция, сопоставляющая объектам какого-то множества числовые значения из ограниченного промежутка.

1.2 Полиномиальное хэширование

Определим строку - как последовательность чисел от 1 до m. Пусть $p = 1e9 + 7$, или же любое другое огромное простое число, а также $k > m$.

Тогда, *прямой полиномиальный хэш строки* есть значение такого многолена:

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

Или же, *обратный полиномиальный хэш*:

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

Сложность. $O(1)$, в таком случае мы поддерживаем переменную, равную нужной в данный момент степени k

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

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

1.3 Количество различных подстрок

Чтобы посчитать количество всех подстрок в строке длины n нужно вычислить хэши всех подстрок и закинуть их в `set`. Тогда, `set.size()` — и будет количеством уникальных посдтрок в строке

Сложность. $O(n^2)$, где n — длина строки, так как всего посдтрок $\dfrac{n(n+1)}{2}$

1.4 Поиск подстроки в строке

Пусть m — длина искомой подстроки. Для начала, вычисляем хэш искомой подстроки, назовем его *эталонным.* После этого идем окном длины m по всей строке, поддерживая хэш текущей подстроки. Если текущий хэш совпал с эталонным, то мы нашли подстроку

Сложность. $O(n)$, так как мы захэшируем всю строку, дл n

1.5 Сравнения подстрок

Для начала, создадим вектор *баз*, который в дальнейшем поможет для полиномиального хэширования строки. Оптимально создать его так

```
int64_t mod = 1e9 + 7;
base[0] = 1;
for (size_t i = 1; i < base.size(); ++i) {
    base[i] = base[i - 1] * 257 % mod;
}
```

Далее вычисляем массив префиксных хэшей, в котором p_i — хэш строки от начала до s_i

```
std::vector<int64_t> p(s.size());
for (size_t i = 1; i < p.size(); ++i) {
    p[i] = (p[i - 1] * base[1] + s[i]) % mod;
}
```

После этого функция, вычисляющая хэш подстрок будет работать так: она принимает два индекса `i` и `j`. После этого вычисляем хэш предыдущей части строки, умноженный на соответствующую базу

```
int64_t hash(size_t i, size_t j){
    int64_t h = p[j] - p[i - 1] * base[j - i + 1] % mod;
    h = (mod + h) % mod;
    return h;
}
```

1.6 Палиндромность подстроки

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

1.7 Количество палиндромов

Можно перебрать центр палиндрома, а для каждого центра — бинпоиском его размер. Далее проверяем подстроку на палиндромность. Заметим, что случаи четных и нечетных палиндромов нужно обрабатывать отдельно

- Перебираем каждый возможный центр (как для нечётных, так и для чётных палиндромов);
- Для каждого центра используем бинарный поиск, чтобы определить максимальный размер палиндрома;
- Проверяем, является ли подстрока палиндромом с помощью хешей;
- Считаем общее количество палиндромов

2 Z-функция. Префикс функция

2.1 Z-функция

Z-функция от строки s — массив z, такой что z_i равно длине максимальной подстроки, начинающейся с i-й позиции, которая равна префиксу s

$$\underbrace{aba}\,c\,\overbrace{aba}\,daba \quad (z_4 = 3)$$

2.2 Построение z-функции за $O(n)$

Будем идти слева направо и хранить *z-блок* — самую 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 символ может лежать либо правее z-блока, либо внутри него:

- Если правее, то мы просто наивно перебором найдем z_i (максимальный отрезок, начинающийся с s_i и равный префиксу), и объявим его новым z-блоком.
- 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 единичку.

```
vector<int> z_function(string s) {
    int n = (int) s.size();
    vector<int> z(n, 0);
    int l = 0, r = 0;
    for (int i = 1; i < n; i++) {
        if (i <= r)
            z[i] = min(r - i + 1, z[i - l]);
        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;
}
```

2.3 Префикс функция

Префикс-функцией от строки s называется массив p, где p_i равно длине самого большого префикса строки $s_0 s_1 s_2 \dots s_i$, который также является и суффиксом i-того префикса (не считая весь i-й префикс)

2.4 Построение префикс функции за $O(n)$

1. **Инициализация**:
    - Создаем массив π длиной N и инициализируем его нулями.
    - Устанавливаем переменную $k = 0$, которая будет отслеживать длину текущего префикса.
2. **Итерация по строке**:
    - Для каждого символа $s[i]$ от 1 до $N - 1$:
        - Если $s[i]$ совпадает с $s[k]$ (т.е. $s[i] == s[k]$):
            * Увеличиваем k на 1 и присваиваем $\pi[i] = k$.

– Если они не совпадают:
  * Пока $k > 0$ и $s[i]$ не совпадает с $s[k]$:
    · Обновляем k с помощью $k = \pi[k-1]$.
  * После выхода из цикла, если $s[i]$ совпадает с $s[k]$, то снова увеличиваем k и присваиваем $\pi[i] = k$.
  * Если нет совпадений, оставляем $\pi[i] = 0$.

2.5 Поиск подстроки в строке

2.5.1 Z-функцией

Пусть у нас есть строка s и паттерн p

- Добавим к строке s символ $\#$

  Можно взять любой другой символ, который гарантированно не встречается ни в p, ни в s и имеет меньший ASCII-код, чем символы строки s. Получим

$$T = \# + s$$

- Теперь вычисляем для строки T её Z-функцию
- После этого ищем в Z-функции все значения равные длине p. Если `z[i]=len(p)`, тогда подстрока p начинается в строке T с позиции i, а значит в строке s — с позиции `i-len(p)-1`

2.5.2 Префикс-функцией

Алгоритм похож на поиск подстроки с применением z-функций

Пусть дана строка t и паттерн s. Составим строку $K = s + \# + t$. Пусть n — длина строки s, а m — строки t

1. Считаем префикс-функцию для строки K
2. Рассмотрим значения префикс-функции, кроме первых $n+1$, так как это строка s и разделитель
   - Если в какой-то позиции i оказалось, что $\pi[i] = n$, то в позиции $i-(n+1)-n+1 = i-2n$ строки t начинается очередное вхождение паттерна

Почему это работает

По определению, значение $\pi[i]$ — самая длинная подстрока, которая заканчивается в позиции i и совпадает с префиксом

В нашем случае, $\pi[i]$ — фактически длина наибольшего блока совпадения со строкой s и оканчивающегося в позиции i

Больше, чем n, эта длина быть не может — за счёт разделителя. Равенство $\pi[i] = n$, означает, что в позиции i оканчивается искомое вхождение строки s

Сложность. $O(n+m)$

2.6 Количество различных подстрок в строке

2.6.1 Z-функцией

Дана строка s длины n. Подсчет уникальных подстрок будем делать итеративно, то есть, зная текущее количество различных подстрок, пересчитывать это количество при добавлении в конец какого-то одного символа.

Пусть k — текущее количество различных подстрок строки s, и мы добавляем в конец символ c

- Возьмём строку $t = s + c$ и инвертируем её. Наша задача — посчитать, сколько у строки t таких префиксов, которые не встречаются в ней более нигде

- Посчитаем Z-функцию для строки t и найдём максимальное значение $z_{\max}$, тогда в строке t встречается (не в начале) её префикс длины $z_{\max}$, но не большей длины. Понятно, префиксы меньшей длины уже точно встречаются в ней

Тогда, число новых подстрок, появляющихся при дописывании символа c, равно $len - z_{\max}$, где len — длина строки **после приписывания символа** c.

Сложность. Для строки длины n — $O(n^2)$

Пример. Допустим, у нас есть строка $abacaba$ и мы хотим приписать в конец символ k. Тогда, посчитаем Z-функцию для строки $kabacaba$:

$$0,0,0,0,0,0,0,0$$

В данном случае, $z_{max} = 0 \Longrightarrow len - z_{max} = 8$, то есть количество «новых» подстрок равно 8

2.6.2 Префикс-функцией

Алгоритм практически идентичен алгоритму определения количества подстрок с помощью z-функции

- Возьмём строку $t = s + c$ и инвертируем её. Наша задача — посчитать, сколько у строки t таких префиксов, которые не встречаются в ней более нигде
- Если мы посчитаем для строки t префикс-функцию и найдём её максимальное значение $\pi_{\max}$, то, очевидно, в строке t встречается (не в начале) её префикс длины $\pi_{\max}$, но не большей длины

Тогда, число новых подстрок, появляющихся при дописывании символа c, равно

$$s.size() + 1 - \pi_{\max}$$

Сложность. Для каждого дописываемого символа мы за $O(n)$ можем пересчитать количество различных подстрок строки. Следовательно, за $O(n^2)$ мы можем найти количество различных подстрок для любой заданной строки

2.7 Сжатие строки

2.7.1 Z-функцией

Дана строка s длины n. Требуется найти самое короткое её «сжатое» представление, т.е. найти такую строку t наименьшей длины, что s можно представить в виде конкатенации одной или нескольких копий t.

Для решения посчитаем Z-функцию строки s, и найдём первую позицию i такую, что $i + z[i] = n$, и при этом n делится на i. Тогда строку s можно сжать до строки длины i.

2.7.2 Префикс-функцией

Проблема в нахождении длины искомой строки t. Зная длину, ответом на задачу будет, например, префикс строки s этой длины.

- Посчитаем по строке s префикс-функцию
- Рассмотрим её последнее значение, т.е. $\pi[n-1]$, и введём обозначение $k = n - \pi[n-1]$
- Если n делится на k, то это k и будет длиной ответа, иначе эффективного сжатия не существует, и ответ равен n

3 Ахо-Корасик

Пусть дан набор строк $s_1, s_2, \ldots, s_m$ алфавита размера k суммарной длины n, называемый *словарем*, и длинный текст t. Необходимо определить, есть ли в тексте хотя бы одно слово из словаря, и если есть, то на какой позиции

3.1 Построение дерева

Пусть у нас есть *бор* — дерево с корнем в некоторой вершине $root$, причём каждое ребро дерева подписано некоторой буквой. При этом, все рёбра, исходящие из некоторой вершины x, должны иметь разные метки

Рассмотрим в боре любой путь из корня; выпишем подряд метки рёбер этого пути. В результате мы получим некоторую строку, которая соответствует этому пути. Если же мы рассмотрим любую вершину бора, то ей поставим в соответствие строку, соответствующую пути из корня до этой вершины.

Каждая вершина бора также имеет флаг **isTerminal**, который равен *true*, если в этой вершине оканчивается какая-либо строка из словаря.

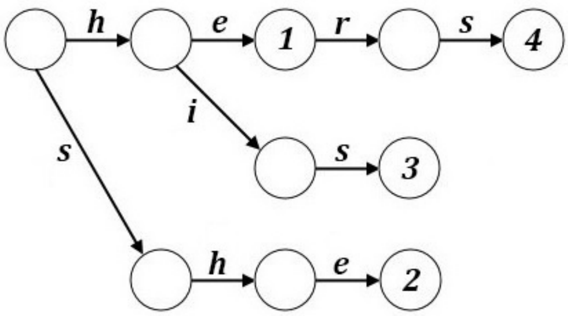
Мы можем хранить бор в виде массива t структур $vertex$. Структура $vertex$ содержит флаг **isTerminal**, и рёбра в виде массива $next$, где $next[i]$ — указатель на вершину, в которую ведёт ребро по символу i, или -1, если такого ребра нет

Вначале бор состоит только из одной вершины — корня, а далее будем добавлять в него строки

Добавление в бор заданной строки s

1. Встаём в корень бора, смотрим, есть ли из корня переход по букве $s[0]$
   - Если переход есть, то переходим по нему в другую вершину
   - Иначе создаём новую вершину и добавляем переход в эту вершину по букве $s[0]$
2. Затем, стоя в какой-то вершине, повторяем процесс для букв $s[1], s[2], \ldots$
3. После окончания процесса помечаем последнюю посещённую вершину, как терминальную

Пример. Пусть у нас есть словарь $S = \{he,\ hers,\ she,\ his\}$, тогда бор для него будет выглядеть так



3.2 Суффиксные и автоматные ссылки

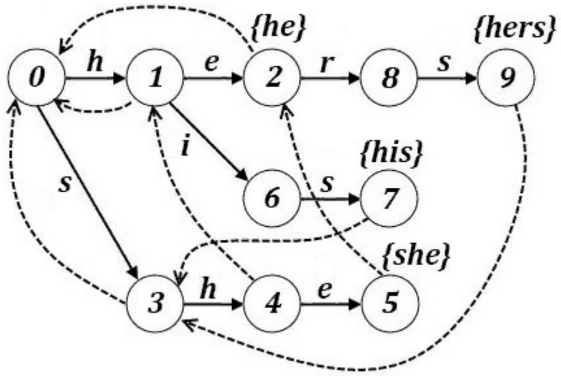
Обозначим за $[u]$ слово, приводящее в вершину u в боре

Определение. Суффиксная ссылка $\pi(u) = v$, если $[v]$ — максимальный суффикс $[u]$, который одновременно является префиксом какого-то слова из словаря, при этом $[v] \neq [u]$

Определение. Автоматный переход $\delta(v, c)$ ведёт в вершину, соответствующую максимальному принимаемому бором суффиксу строки $v + c$.

3.2.1 Построение суффиксных ссылок

Суффиксная ссылка для каждой вершины u — это вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине u. Единственный особый случай — корень бора: для удобства суффиксную ссылку из него проведём в себя же.



Например, для вершины 5 с соответствующей ей строкой she максимальным подходящим суффиксом является строка he. Видим, что такая строка заканчивается в вершине 2. Следовательно суффиксной ссылкой вершины для 5 является вершина 2

3.2.2 Вычисление автоматных ссылок

Автоматные ссылки вычисляются с использованием *суффиксных ссылок* и обычных переходов. Алгоритм строит автоматные ссылки по следующей логике:

1. Мы стоим в некоторой вершине v и хотим перейти по символу c, тогда
   - Если есть обычный переход по символу c, то автоматная ссылка просто указывает на этот переход: `v->autLink[c] = &v->next[c]`
   - Если из вершины v нет перехода по c, то автоматная ссылка переходит по суффиксной ссылке (если ее нет, то автомат идет в корень)
     - Смотрим, есть ли переход по символу c из вершины, в которую мы попали
     - Если да, то автоматная ссылка указывает на вершину, в которую мы попадем, сделав переход по c
     - Если нет, то снова пытаемся пройти по суффиксной ссылке, повторяя алгоритм, пока не попадем либо в корень, либо к нужному ребру
   - Если текущая вершина — корень, то автоматная ссылка замыкается на сам корень, то есть: `v->autLink[c] = v`
   - Если это не корень, то автоматная ссылка указывает на результат перехода по суффиксной ссылке: `v->autLink[c] = v->sufLink->autLink[c]`

3.2.3 Работа автоматных ссылок и зачем они нужны

Предположим, что мы находимся в вершине v и хотим обработать символ c. Если в текущей вершине v есть переход по символу c, то мы идем по обычному переходу (ребру), и всё продолжается как обычно.

Фактически, автоматная ссылка — это резервный переход в другую вершину, где мы можем продолжить поиск.

Автоматные ссылки помогают избежать возврата назад по тексту при поиске. Если символ не подошел, мы просто переходим к другой вершине, которая может соответствовать суффиксу уже обработанной части текста

Сложность. После построения автоматных ссылок мы можем переходить по ним за $O(1)$

3.2.4 Сложность

Пусть у нас есть словарь $s_1, s_2, \ldots, s_n$

Построение бора, суффиксных ссылок и автомата займет $O(m)$, где $m = \sum\limits_{i=1}^{n} |s_i|$

1. В боре каждая вершина соответствует одному символу из набора строк. Значит, мы добавляем каждый символ всех строк в бор один раз

2. Суффиксные ссылки для каждой вершины бора строятся с помощью BFS. Для каждой вершины мы проверяем её суффиксную ссылку, а это делается за $O(m)$, так как каждая вершина обрабатывается один раз

3. Автоматные ссылки тоже строятся в процессе обхода бора по тому же принципу, что и суффиксные ссылки. Для каждой вершины выполняется один проход по суффиксной ссылке и проверяются символы, что снова занимает $O(m)$ времени (построение)

4 Суффиксный массив

Суффиксным массивом строки s называется перестановка индексов начал её суффиксов, которая задаёт порядок их лексикографической сортировки. Иными словами, чтобы его построить, нужно выполнить сортировку всех суффиксов заданной строки

SA		**SA**	
1	mississippi$	12	$
2	ississippi$	11	i$
3	ssissippi$	10	ippi$
4	sissippi$	9	ppi$
5	issippi$	8	pi$
6	ssippi$	7	sippi$
7	sippi$	6	ssippi$
8	ippi$	5	issippi$
9	ppi$	4	sissippi$
10	pi$	3	ssissippi$
11	i$	2	ississippi$
12	$	1	mississippi$

Сортировка всех суффиксов строки «mississippi$»

4.1 Сортировка суффиксов

Можно выделить три основных способа отсортировать суфмасс:

- Пишем компаратор, который сравнивает все суффиксы, кидаем это в `std::sort`. Сложность — $O(n^2 \log n)$
- Используем хэшами, тогда асимптотика — $O(n \log^2 n)$
- Самый оптимальный метод — $O(n \log n)$

Рассмотрим самый оптимальный метод. В процессе мы будем использовать ранг суффикса — метка о лексикографическом порядке суффикса.

1. Инициализация рангов
   - Каждый суффикс строки сортируется по его первому символу. Это можно сделать с помощью сортирови подсчетом
   - Каждый символ строки преобразуется в числовой ранг на основе его порядкового номера в алфавите
2. Итеративная сортировка по подстрокам длины 2^k
   - На каждом шаге мы удваиваем длину подстрок, по которым сортируем суффиксы
   - Сначала сортируем по первым символам ($k = 1$), затем по первым двум символам ($k = 2$), четырём ($k = 4$), и так далее, пока длина подстроки не станет больше длины строки
3. На каждом шаге сортировка суффиксов по подстрокам длины 2^k происходит в два этапа:
   - Сортировка по первым k символам (используя ранги из предыдущего шага)
   - Сортировка по следующим k символам (если они есть)
4. Обновление рангов
   - После сортировки нужно обновить ранги суффиксов на основе их позиций в отсортированном массиве
   - Если два суффикса равны по первым k символам, им присваивается одинаковый ранг. Если не равны, их ранги различаются

Процесс продолжается, пока длина подстрок, по которым происходит сортировка, не станет больше длины строки

Сложность. Этот алгоритм работает за $O(n \log n)$, где n — длина строки, поскольку на каждом этапе сортировки подстрок мы используем сортировку подсчётом, которая работает за линейное время, и таких этапов будет примерно $\log n$, потому что мы каждый раз удваиваем длину строки

4.2 Применение в задачах

Суффиксный массив применяется в таких задачах, как

- Поиск подстроки в строке
- Подсчет количества различных подстрок
- Нахождение LCP двух строк

4.3 Поиск LCP. Алгоритм пяти корейцев

Мы построили суффиксный массив, теперь попробуем найти LCP **Сложность.** $O(n)$, где n — длина строки

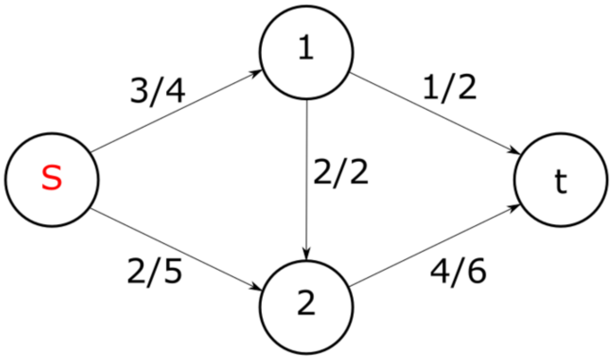
Алгоритм работает следующим образом

1. Построение массива рангов:
   - Для каждой позиции строки вычисляется её ранг, то есть позиция соответствующего суффикса в суффиксном массиве
   - Пусть `rank[i]` — это индекс позиции i в суффиксном массиве
2. Итерирование по суффиксам
   - Для каждого суффикса строки на позиции i мы пытаемся найти длину наибольшего общего префикса между суффиксом на позиции i и суффиксом, который идёт перед ним в лексикографическом порядке
3. Если LCP для предыдущих суффиксов уже найден, можно использовать его для ускорения поиска. Мы знаем, что суффиксы уже совпадают на определённой длине и можем продолжать с этой длины)

```
vector<int> calc_lcp(vector<int> &val, vector<int> &c, vector<int> &p) {
    int n = val.size();
    int l = 0;
    vector<int> lcp(n);
    for (int i = 0; i < n; i++) {
        if (c[i] == n - 1)
            continue;
        int nxt = p[c[i] + 1];
        while (max(i, nxt) + l < n && val[i + l] == val[nxt + l])
            l++;
        lcp[c[i]] = l;
        l = max(0, l - 1);
    }
    return lcp;
}
```

5 Задача построения максимального потока в сети

Задача заключается в том, чтобы найти максимальный поток, который можно провести из истока в сток через сеть, состоящую из вершин и ориентированных рёбер с заданными пропускными способностями



5.1 Алгоритм Форда-Фалкерсона

Алгоритм основан на методе поиска увеличивающих путей в сети

Определение. *Увеличивающий путь* — это путь от источника к стоку, по которому можно увеличить поток, не превышая пропускные способности рёбер

1. Установить начальный поток в сети равным нулю
2. Поиск увеличивающего пути
   - Для этого можно использовать любой алгоритм поиска (DFS или BFS) для нахождения пути, где все рёбра имеют положительную остаточную пропускную способность
3. После нахождения увеличивающего пути определить минимальную остаточную пропускную способность по этому пути. Эта величина обозначает, насколько можно увеличить поток
4. Увеличить поток
   - Увеличить поток по всем рёбрам увеличивающего пути на значение минимальной остаточной пропускной способности
   - Уменьшить остаточную пропускную способность по всем рёбрам пути на это значение
   - Увеличить остаточную пропускную способность обратных рёбер (если поток проходит по ребру, создаётся обратное ребро, через которое поток может быть уменьшен в будущем)

Как только не удается найти увеличивающий путь, алгоритм завершает работу

Сложность. $O(|E|f)$, где E — число рёбер в графе, f — максимальный поток в графе, так как каждый увеличивающий путь может быть найден за $O(E)$ и увеличивает поток как минимум на 1

5.2 Минимальный разрез сети

Определение. *Минимальный разрез сети* — это разделение графа на две части (одна часть содержит исток, другая — сток), которое минимизирует сумму пропускных способностей рёбер, пересекающих разрез в одном направлении

Примечание. Величина максимального потока равна пропускной способности минимального разреза

5.3 Алгоритм Эдмондса-Карпа

Алгоритм Эдмонса-Карпа — это улучшение алгоритма Форда-Фалкерсона, которое использует поиск в ширину (BFS) для нахождения увеличивающих путей в графе. Этот алгоритм решает задачу нахождения максимального потока в сети с гораздо более предсказуемым временем работы, чем оригинальный алгоритм Форда-Фалкерсона

1. Положим все потоки равными нулю. Остаточная сеть изначально совпадает с исходной сетью
2. В остаточной сети находим кратчайший путь из источника в сток. Если такого пути нет, останавливаемся
3. Пускаем через найденный путь (он называется увеличивающим путём или увеличивающей цепью) максимально возможный поток
    - На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью $c_{\min}$
    - Для каждого ребра на найденном пути увеличиваем поток на $c_{\min}$, а в противоположном ему — уменьшаем на $c_{\min}$
    - Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных им рёбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его

Сложность. $O(VE^2)$, где V — количество вершин в графе, а E — количество рёбер

6 Максимальное паросочетание в двудольном графе

В задаче поиска максимального паросочетания в графе требуется найти наибольший набор рёбер, не имеющих общих вершин

Определение. *Паросочетание* — это набор рёбер, не имеющих общих вершин, то есть никакие два ребра не должны делить вершину

6.1 Алгоритм Куна

Начнем с пустого паросочетания и будем искать увеличивающие цепи, пока они ищутся

1. Для каждой вершины из множества U пытаемся найти её пару из множества V
2. Ищем увеличивающие пути
   - Для каждой вершины $u \in U$ проверяем, можно ли её присоединить к какому-то элементу из множества V. Используем для этого DFS
   - Если вершина не используется в другом паросочетании, то соединяем, если используется, запускаем DFS для вершины из V и пытаемся присоединить к другой вершине
   - Если удаётся найти увеличивающий путь (путь, по которому можно провести обмен, чтобы увеличить паросочетание), то обновляем текущее паросочетание
3. Каждый раз, когда находим увеличивающий путь, добавляем его в текущее паросочетание и пересчитываем.

Сложность. $O(VE)$, где E — количество рёбер, а V — количество вершин

7 Деревья поиска

Бинарное дерево поиска — дерево, для которого выполняются следующие свойства:

- У каждой вершины не более двух детей
- Все вершины обладают *ключами*, на которых определена операция сравнения (например, целые числа или строки)
- У всех вершин *левого* поддерева вершины v ключи *не больше*, чем ключ v
- У всех вершин *правого* поддерева вершины v ключи *больше*, чем ключ v
- Оба поддерева — левое и правое — являются двоичными деревьями поиска

В *небинарных* деревьях количество детей может быть больше двух, и при этом в «более левых» поддеревьях ключи должны быть меньше, чем «более правых»

Для работы с деревьями поиска нужно создать структуру

```
struct Node:
  T key                    // key of the node
  Node left                // pointer to the left child
  Node right               // pointer to the right child
  Node parent              // pointer to the parent
```

7.1 Поиск элемента

Нужна функция, принимающая корень дерева и искомый ключ

- Для каждого узла сравниваем значение его ключа с искомым ключом
- Если ключи одинаковы, то функция возвращает текущий узел
- В противном случае функция вызывается рекурсивно для левого или правого поддерева

Сложность в худшем случае — $O(h)$ (h — высота дерева), так как узлы, которые посещает функция образуют нисходящее дерево. Такое возможно, когда дерево является «бамбуком»

Сложность при оптимизации — $O(\log N)$. Если изменить способ хранения дерева, например сразу при проходе до какого-то ключа записать его как ключ ко всем вершинам в пути, то сложность снизится

7.2 Вставка элемента

Почти то же самое, что поиск элемента, но теперь при обнаружении у элемента отсутствия ребенка нужно подвесить на него вставляемый элемент

```
Node insert(x : Node, z : T):          // x - root of the subtree, z - key to be inserted
   if x == null
      return Node(z)                   // attach a Node with key = z
   else if z < x.key
      x.left = insert(x.left, z)
   else if z > x.key
      x.right = insert(x.right, z)
   return x
```

7.3 Удаление элемента

Рассмотрим три случая при рекурсивной реализации

1. Удаляемый элемент находится в *левом* поддереве текущего поддерева

   - тогда нужно рекурсивно удалить элемент из нужного поддерева

2. Удаляемый элемент находится в *правом* поддереве
   - тогда нужно рекурсивно удалить элемент из нужного поддерева

3. Удаляемый элемент находится в *корне*, то два случая:
   - имеет два дочерних узла
     - нужно заменить его минимальным элементом из правого поддерева и рекурсивно удалить этот минимальный элемент из правого поддерева
   - имеет один дочерний узел
     - нужно заменить удаляемый элемент потомком

7.4 AVL-деревья

Определение. *AVL-дерево* — сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её двух поддеревьев различается не более чем на 1

Определение. *Баланс дерева* — это разница между высотой левого и правого поддерева

7.5 Добавление элемента и балансировка AVL-дерева

1. **Добавление элемента**
   - Используем стандартный алгоритм добавления в бинарное дерево поиска (BST). Сравниваем значение добавляемого элемента с текущим узлом:
     - Если значение меньше текущего, переходим к левому поддереву.
     - Если больше — к правому.
   - После нахождения подходящей позиции вставляем новый узел.

2. **Обновление высот**
   - После добавления узла необходимо обновить высоты всех предков вставленного узла. Для каждого узла, начиная с родителя вставленного узла до корня, вычисляем новую высоту как `1 + max(height(left), height(right))`

3. **Проверка баланса**
   - Для каждого узла, начиная с родителя вставленного узла, проверяем его баланс:
   - Если баланс узла стал равен 2 или -2, значит, дерево стало несбалансированным, и нужно выполнить ротации.

4. **Повороты**
   - **LL-rotation (Правый поворот):** Если баланс 2, а добавление произошло в левое поддерево левого дочернего узла.
   - **RR-rotation (Левый поворот):** Если баланс -2, а добавление произошло в правое поддерево правого дочернего узла.
   - **LR-rotation (Левый-правый поворот):** Если баланс 2, а добавление произошло в правое поддерево левого дочернего узла. Сначала выполняем левый поворот на левом дочернем узле, затем правый поворот на текущем узле.
   - **RL-rotation (Правый-левый поворот):** Если баланс -2, а добавление произошло в левое поддерево правого дочернего узла. Сначала выполняем правый поворот на правом дочернем узле, затем левый поворот на текущем узле.

7.6 Высота AVL-дерева

AVL-дерево с n ключами имеет высоту $O(\log n)$