

Алгоритмы и структуры данных

Экзамен, 2-й семестр

1 Медиана и порядковые статистики. Алгоритм с линейным временем работы для медианы

k -я порядковая статистика набора элементов линейно упорядоченного множества — такой его элемент, который является k -м элементом набора в порядке сортировки

Медиана — k -я порядковая статистика при $k = N/2$ (если N не кратно двум, то будем рассматривать нижнюю медиану $k = \lfloor N/2 \rfloor$ и верхнюю медиану $k = \lceil N/2 \rceil$)

Алгоритм с линейным временем работы Идея алгоритма напоминает QuickSort.

Сам алгоритм:

1. Все n элементов входного массива разбиваются на группы по пять элементов, в последней группе будет $n \bmod 5$ элементов. Эта группа может оказаться пустой при n кратным 5.
2. Сначала сортируется каждая группа, затем из каждой группы выбирается медиана.
3. Путем рекурсивного вызова шага определяется медиана x из множества медиан (верхняя медиана в случае чётного количества), найденных на втором шаге. Найденный элемент массива x используется как рассекающий (за i обозначим его индекс).
4. Массив делится относительно рассекающего элемента x .
5. Если $i = k$, то возвращается значение x . Иначе запускается рекурсивно поиск элемента в одной из частей массива: k -ой статистики в левой части при $i > k$ или $(k - i - 1)$ -ой статистики в правой части при $i < k$

Доказательство оценки сложности:

Сначала определим нижнюю границу для количества элементов, превышающих по величине рассекающий элемент x . В общем случае как минимум половина медиан, найденных на втором шаге, больше или равны медиане медиан x . Таким образом, как минимум $\frac{n}{10}$ групп содержат по 3 превышающих величину x , за исключением группы, в которой меньше 5 элементов и ещё одной группы, содержащей сам элемент x . Таким образом получаем, что количество элементов больших x , не менее $\frac{3n}{10}$.

Проведя аналогичные рассуждения для элементов, которые меньше по величине, чем рассекающий элемент x , мы получим, что как минимум $\frac{3n}{10}$ меньше, чем элемент x .

Само время работы $T(n)$ не меньше, чем

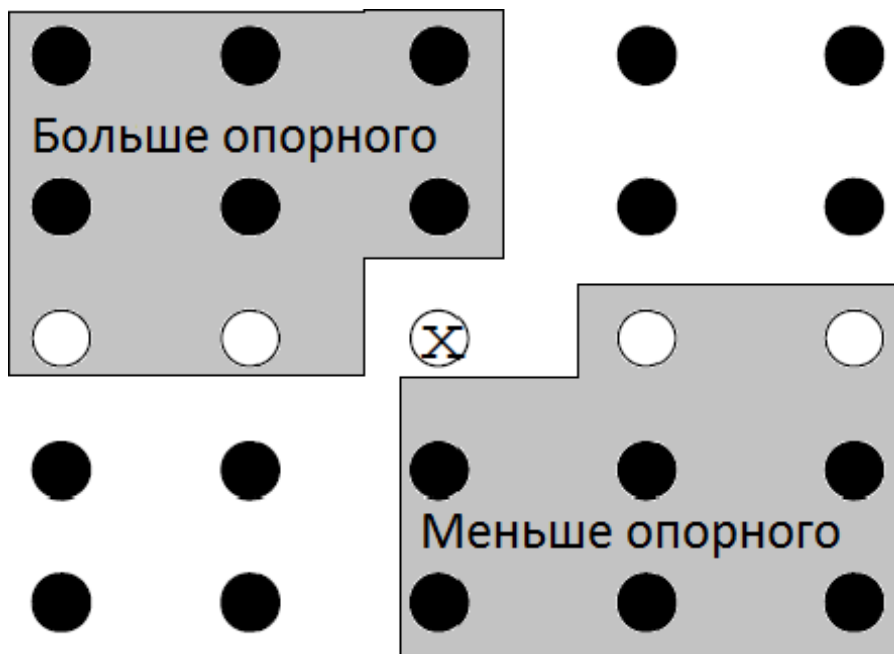


Иллюстрация к рассуждению выше

1. Время работы на сортировку группы (одна группа сортируется за константное время, так как в каждой группе константное количество элементов) и разбиение по рассекающему элементу (аналогично операции merge) — $O(n)$
2. времени работы для поиска медианы медиан, то есть $T\left(\frac{n}{5}\right)$
3. времени работы для поиска k -го элемента в одной из двух частей массива, то есть $T(s)$, где s — количество элементов в этой части. Но s не превосходит $\frac{7n}{10}$, так как чисел, меньших рассекающего элемента, не менее $\frac{3n}{10}$ — это $\frac{n}{10}$ медиан, меньших медианы медиан, плюс не менее $2n/10$ элементов, меньших этих медиан. С другой стороны, чисел, больших рассекающего элемента, так же не менее $\frac{3n}{10}$, следовательно, $s \leq \frac{7n}{10}$, то есть в худшем случае $s = \frac{7n}{10}$.

Итоговое время работы:

$$T(n) \leq T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + Cn$$

Докажем по индукции, что $T(n) \leq 10Cn$

Предположим, что наше неравенство $T(n) \leq 10Cn$ выполняется при малых n , для некоторой достаточно большой константы C . Тогда, по предположению индукции,

$$T(n) \leq T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + Cn \leq 10C \cdot \frac{7n}{10} + 10C \cdot \frac{n}{5} + Cn = 10Cn$$

Значит, итоговая сложность алгоритма $O(n)$

2 Сортировка массива: пузырьковая, mergesort, quicksort. Алгоритмы и оценки сложности.

Пузырьковая сортировка:

Сложность: $O(n^2)$

Алгоритм: несколько раз проходимся по массиву, «поднимая» элемент в начало.

Mergesort (слияние)

Сложность: $T(n) = 2T(\frac{n}{2}) + O(n)$ или же $O(n \log n)$

Алгоритм: Разбиваем наш массив на 2, пока не останется по одному элементу, затем рекурсивно сравниваем каждый элемент с соседним, сортируем, объединяем.

Quicksort

Сложность в худшем случае — $O(n^2)$, в среднем — $O(n \log n)$

Алгоритм:

1. Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но может зависеть его эффективность.
2. Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньше опорного», «равные», и «большие».
3. Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

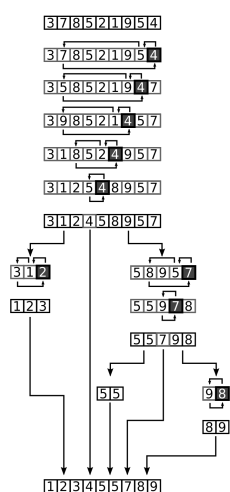


Figure 2: Quicksort

3 Списки: односвязный и двусвязный

4 Бинарные деревья поиска. Вставка, удаление, оценки сложности.

5 Графы. Способы их представления в памяти компьютера. Матрицы смежности, матрицы инцидентности, списки связности, представление на двух массивах (CSR).

6 Обходы графов. Обход в ширину, обход в глубину.

7 Алгоритмы поиска кратчайших путей. Алгоритм Дейкстры, алгоритм Беллмана-Форда.

8 Остовные деревья. Алгоритмы Прима, Краскала, Борувки.

9 Эвристики для поиска кратчайших путей, алгоритм A^* .

10 Потоки в сетях. Максимальный поток и минимальный разрез.]

См билет 6 из сложных

11 Хэш-функции. Коллизии. Хеш-таблицы.
Хэширование. Фильтр Блюма.

12 Предикат поворота. Задача пересечения двух отрезков

13 Выпуклые оболочки, алгоритма Джарвиса, Грэхема и QuickHull.

14 kD-деревья. Окто- и квадро-деревья.

15	Многочлены. Карацубы.	Метод Горнера.	Умножение
----	--------------------------	----------------	-----------

1 Основная теорема для рекуррентных соотношений. Схема доказательства.

Теорема

Пусть $T(n) = a \cdot T(\lceil n/b \rceil) + O(n^d)$. Тогда

$$T(n) = \begin{cases} O(n^d), & d > \log_b a \\ O(n^d \log n), & d = \log_b a, \\ O(n^{\log_b a}), & d < \log_b a \end{cases}$$

где $a \geq 1$ — количество частей, на которые мы дробим задачу, $b > 1$ — во сколько раз легче становится решить задачу, d — степень сложности входных данных.

Схема доказательства:

Рассмотрим дерево рекурсии данного соотношения. Всего в нем будет $\log_b n$ уровней. На каждом таком уровне, количество детей в дереве будет умножаться на a на уровне i будет a^i потомков. Также известно, что каждый ребенок на уровне i размера $\frac{n}{b^i}$. Ребенок размера $\frac{n}{b^i}$ требует $O\left(\left(\frac{n}{b^i}\right)^d\right)$ дополнительных затрат, поэтому общее количество совершенных действий на уровне i :

$$O\left(a^i \cdot \left(\frac{n}{b^i}\right)^d\right) = O\left(n^d \cdot \left(\frac{a^i}{b^{id}}\right)\right) = O\left(n^d \cdot \left(\frac{a}{b^d}\right)^i\right)$$

Решение разбивается на три случая: когда $\frac{a}{b^d}$ больше 1, равна 1 или меньше 1. Переход между этими случаями осуществляется при

$$\frac{a}{b^d} = 1 \Leftrightarrow a = b^d \Leftrightarrow \log_b a = d \cdot \log_b b \Leftrightarrow \log_b a = d$$

Распишем всю работу в течение рекурсивного спуска:

$$T(n) = \sum_{i=0}^{\log_b n} O\left(n^d \left(\frac{a}{b^d}\right)^i\right) + O(1) = O\left(n^d \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i\right)$$

Отсюда получаем:

1. $d > \log_b a \Rightarrow T(n) = O(n^d)$ (так как $\left(\frac{a}{b^d}\right)^i$ — бесконечно убывающая геометрическая прогрессия)
2. $d = \log_b a \Rightarrow T(n) = O\left(n^d \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i\right) =$
 $= O\left(n^d \cdot \sum_{i=0}^{\log_b n} (1)^i\right) = O(n^d + n^d \cdot \log_b n) = O(n^d \cdot \log_b n)$

$$3. \ d < \log_b a \Rightarrow T(n) = O\left(n^d \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i\right) = O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right), \text{ HO}$$

$$n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \cdot \left(\frac{a^{\log_b n}}{b^{d \log_b n}}\right) = n^d \cdot \left(\frac{n^{\log_b a}}{n^d}\right) = n^{\log_b a} \Rightarrow T(n) = O(n^{\log_b a})$$

2 АВЛ-деревья. Повороты, балансировка.

АВЛ-дерево - сбалансированное двоичное дерево поиска со следующим свойством: для каждой его вершины высота её двух поддеревьев различается не более чем на 1. Названо в честь изобретателей Г. М. Адельсона-Вельского и Е. М. Ландиса.

Теорема: АВЛ-дерево имеет высоту $h = O(\log n)$.

Доказательство: Высоту поддерева с корнем x будем обозначать как $h(x)$. Пусть m_h - минимальное число вершин в АВЛ-дереве высоты h . Тогда легко видеть, что $m_{h+2} = m_{h+1} + m_h + 1$ по индукции. Равенством $m_h = F_{h+2} - 1$ докажем.

База: $m_1 = F_3 - 1$ - верно, $m_1 = 1$, $F_3 = 2$.

Шаг: Допустим $m_h = F_{h+2} - 1$ - верно. Тогда $m_{h+1} = m_h + m_{h-1} + 1 = F_{h+2} - 1 + F_{h+1} - 1 + 1 = F_{h+3} - 1$.

$F_h = \Omega(\varphi^h)$ где $\varphi = \frac{\sqrt{5}+1}{2}$. То есть $n \geq \varphi^h \Rightarrow \log_\varphi n \geq h$. Высота АВЛ-дерева из n вершин - $O(\log n)$.

Балансировка: Балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев $|h(L) - h(R)| = 2$ изменяет связи предок - потомок в поддереве данной вершины так, чтобы восстановилось свойство дерева $|h(L) - h(R)| = 1$, иначе ничего не меняет. ($diff[i] = h(L) - h(R)$).

Малое вращение: ($O(1)$)

- Левое используется когда $h(b) - h(L) = 2$ и $h(c) \leq h(R)$.
- Правое используется, когда $h(a) - h(R) = 2$ и $h(c) \leq h(L)$.

Большое вращение: ($O(1)$) Левое используется когда $h(b) - h(L) = 2$ и $h(c) > h(R)$. Правое используется когда $h(b) - h(R) = 2$ и $h(c) > h(L)$.

Большое вращение состоит из двух малых.

Вставка (insert): Спускаемся по дереву, как при поиске. Если мы стоим в вершине a и там надо идти в поддерево b , то делаем b листом, а вершину a корнем. Поднимаемся вверх по пути поиска и пересчитываем баланс у вершин. Если мы поднялись в вершину i из левого поддерева, то $diff[i] = h(L) - h(R)$ увеличилось на 1. Если из правого - уменьшается на 1. Если пришли в вершину и баланс стал равен 0, то высота не изменилась и подъём останавливается. Если пришли в вершину и её баланс стал равен 1 или -1, то высота поддерева изменилась и подъём продолжается. Если пришли в вершину и её баланс стал равен 2 или -2, то делаем одно из 4 вращений и, если после вращения баланс стал 0, то останавливаемся, иначе продолжаем подъём. Сложность: $O(\log n)$, т.к. в процессе добавления вершины мы рассматриваем не более $O(\log n)$ вершин, и для каждой запускаем балансировку не более одного раза.

Удаление (delete): Если вершина лист, удаляем её. Иначе найдём самую близкую по значению вершину и, переместим её на место удаляемой вершины, а затем удалим эту вершину. От удалённой вершины будем подниматься к корню и пересчитывать баланс у вершин. $diff[]$ уменьшается. Если поднялись из левого поддерева, то $diff[]$ уменьшается на 1. Если из правого - увеличивается на 1. Если пришли в вершину и её баланс стал равен 1 или -1, то высота поддерева не изменилась и подъём можно остановить. Если пришли в вершину и баланс стал равен 0, то высота поддерева уменьшилась и подъём нужно продолжить. Если пришли в вершину и её баланс стал равен 2 или -2, то делаем одно из 4 вращений и, если после вращения баланс стал 0, то подъём продолжается, иначе - останавливается. Аналогично с вставкой: $O(\log n)$.

Поиск (find): Как в обычном бинарном дереве поиска. $O(\log n)$.

3 Красно-черные деревья. Балансировка (схема).

Опр. 3.1. Красно-черное дерево — бинарное дерево поиска, у которого каждому узлу сопоставлен дополнительный атрибут — цвет и для которого выполняются следующие свойства:

1. Каждый узел либо красный, либо черный.
2. Корень дерева является черным узлом.
3. Каждый лист дерева (NULL) является черным узлом.
4. Если узел красный, то оба его дочерних узла черные.
5. Для каждого узла все простые пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов.

Опр. 3.2. Чёрной высотой $bh(x)$ вершины x называется количество черных узлов на любом простом пути от узла x (не считая сам узел) к листу. В соответствии со свойством 5 красно-черных деревьев черная высота узла — точно определяемое значение, поскольку все нисходящие простые пути из узла содержат одно и то же количество черных узлов. Чёрной высотой дерева считается черную высоту его корня.

Лемма 3.1. Красно-черное дерево с n внутренними узлами имеет высоту, не превышающую $2 \log_2(n + 1)$.

Доказательство. Покажем, что поддереву любого узла x содержит как минимум $2^{bh(x)} - 1$ внутренних узлов. Докажем это по индукции по высоте x . Если высота x равна 0, то узел x должен быть листом (NULL), а поддерево узла x содержит не менее $2^{bh(x)} - 1 = 2^0 - 1 = 0$ внутренних узлов. Теперь для выполнения шага индукции рассмотрим узел x , который имеет положительную высоту и представляет собой внутренний узел с двумя потомками. Каждый дочерний узел имеет черную высоту либо $bh(x)$, либо $bh(x) - 1$ в зависимости от того, является ли его цвет соответственно красным или черным. Поскольку высота потомка x меньше, чем высота самого узла x , мы можем использовать предположение индукции и сделать вывод о том, что каждый из потомков x имеет как минимум $2^{bh(x)-1} - 1$ внутренних узлов. Таким образом, дерево с корнем в вершине x содержит как минимум $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ внутренних узлов, что и доказывает наше утверждение.

Для того чтобы завершить доказательство леммы, обозначим высоту дерева через h . Согласно свойству 4 по крайней мере половина узлов на

любом простом пути от корня к листу, не считая сам корень, должны быть черными. Следовательно, черная высота корня должна составлять как минимум $h/2$; значит,

$$n \geq 2^{h/2} - 1$$

. Переносим 1 в левую часть и логарифмируя, получим, что $\log_2(n+1) \geq h/2$, или $h \leq 2 \log_2(n+1)$.

Непосредственным следствием леммы является то, что такие операции над динамическими множествами, как Search, Minimum, Maximum, Predecessor и Successor, при использовании красно-черных деревьев выполняются за время $O(\log h)$, поскольку время работы этих операций на дереве поиска высотой h составляет $O(h)$, а любое красно-черное дерево с n узлами является деревом поиска высотой $O(\log n)$.

3.1 Операции

3.1.1 Вставка

По умолчанию производится вставка красной вершины. Если её предок чёрный, всё в порядке; иначе возможны следующие варианты:

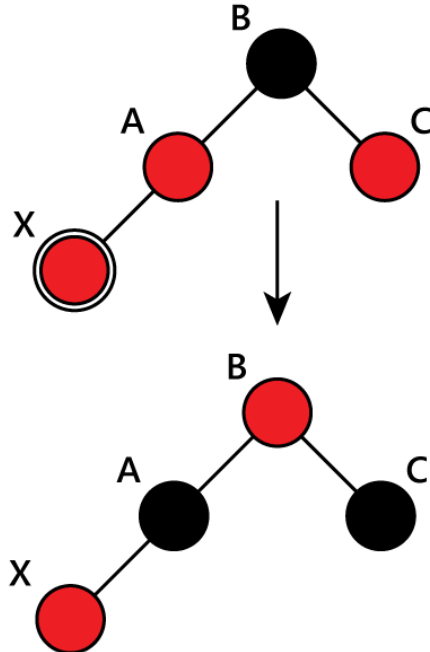
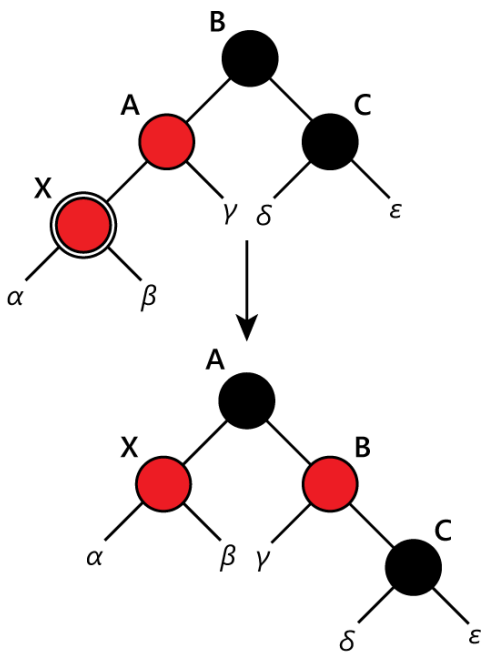
3.1.2 Удаление

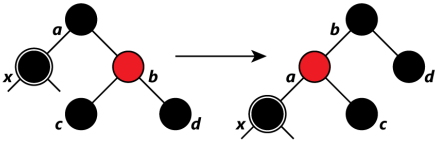
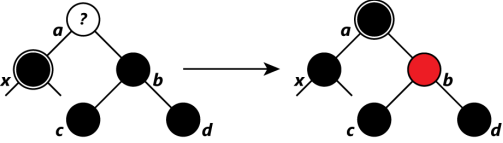
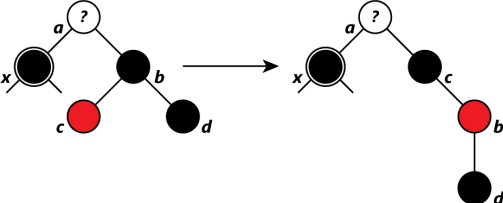
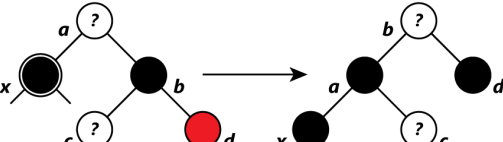
При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

1. Если у вершины нет детей, то изменяем указатель на неё у родителя на NULL.
2. Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.
3. Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка (так как такая вершина находится в правом поддереве исходной вершины и она самая левая в нём, иначе бы мы взяли её левого ребенка. Иными словами, сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый ребенок). Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева.

«Дядя» красный	«Дядя» черный
<p>Перекрашиваем «отца» и «дядю» в чёрный цвет, а «деда» — в красный. Черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин «отцы» черные. Проходом вверх до корня проверяем, не нарушена ли балансировка. Не забываем, что корень всегда черный.</p>	<p>Выполняем поворот, затем перекрашивание. Если добавляемый узел был правым потомком, то вращение — левое, а узел становится левым потомком.</p>
	

<p>Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас x имеет чёрного брата и красного отца. Таким образом, мы можем перейти к одному из следующих случаев.</p>	
<p>Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество чёрных узлов на путях, проходящих через b, но добавит один к числу чёрных узлов на путях, проходящих через x, восстанавливая тем самым влияние удаленного чёрного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.</p>	
<p>Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x, ни его отец не влияют на эту трансформацию.</p>	
<p>Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойства 3 и 4 не нарушаются. Но у x теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.</p>	

4 Бинарные кучи. Реализация с указателями и на массиве. Добавление и удаление элемента в бинарную кучу.

Опр. 4.1. Двоичная куча или пирамида (англ. Binary heap) — такое двоичное дерево, для которого выполнены следующие три условия:

1. Значение в любой вершине не больше (если куча для минимума), чем значения в её потомках.
2. На i -м слое (кроме, может быть, последнего) 2^i вершин. Слои нумеруются с нуля.
3. Последний слой заполнен слева направо.

Удобнее всего двоичную кучу хранить в виде массива $a[0..n-1]$, у которого нулевой элемент, $a[0]$ — элемент в корне, а потомками элемента $a[i]$ являются $a[2i + 1]$ и $a[2i + 2]$. Высота кучи определяется как высота двоичного дерева. То есть она равна количеству рёбер в самом длинном простом пути, соединяющем корень кучи с одним из её листьев. Высота кучи есть $O(\log n)$, где n — количество узлов дерева.

Чаще всего используют кучи для минимума (когда предок не больше детей) и для максимума (когда предок не меньше детей).

Двоичные кучи используют, например, для того, чтобы извлекать минимум из набора чисел за $O(\log n)$. Они являются частным случаем приоритетных очередей.

4.1 Восстановление свойств кучи

Если в куче изменяется один из элементов, то она может перестать удовлетворять свойству упорядоченности. Для восстановления этого свойства служат процедуры `siftDown` (просеивание вниз) и `siftUp` (просеивание вверх).

4.1.1 `siftDown`

Если значение измененного элемента увеличивается, то свойства кучи восстанавливаются функцией `siftDown`.

Работа процедуры: если i -й элемент меньше, чем его сыновья, всё поддерево уже является кучей, и делать ничего не надо. В противном случае меняем местами i -й элемент с наименьшим из его сыновей, после чего выполняем `siftDown` для этого сына. Процедура выполняется за время $O(\log n)$.

```

function siftDown(i : int):
    while 2 * i + 1 < a.heapSize      // heapSize - количество элементов в
        left = 2 * i + 1              // left - левый сын
        right = 2 * i + 2             // right - правый сын
        j = left
        if right < a.heapSize and a[right] < a[left]
            j = right
        if a[i] <= a[j]
            break
        swap(a[i], a[j])
        i = j

```

4.1.2 siftUp

Если значение измененного элемента уменьшается, то свойства кучи восстанавливаются функцией siftUp.

Работа процедуры: если элемент больше своего отца, условие 1 соблюдено для всего дерева, и больше ничего делать не нужно. В противном случае мы меняем местами его с отцом, после чего выполняем siftUp для этого отца. Иными словами, слишком маленький элемент всплывает вверх. Процедура выполняется за время $O(\log n)$.

```

function siftUp(i : int):
    while a[i] < a[(i - 1) / 2]      // i = 0 - мы в корне
        swap(a[i], a[(i - 1) / 2])
        i = (i - 1) / 2

```

4.2 Извлечение минимального элемента

Выполняет извлечение минимального элемента из кучи за время $O(\log n)$. Извлечение выполняется в четыре этапа:

1. Значение корневого элемента (он и является минимальным) сохраняется для последующего возврата.
2. Последний элемент копируется в корень, после чего удаляется из кучи.
3. Вызывается siftDown для корня.
4. Сохранённый элемент возвращается.

```

int extractMin():
    int min = a[0]
    a[0] = a[a.heapSize - 1]
    a.heapSize = a.heapSize - 1

```

```
siftDown(0)
return min
```

4.3 Добавление нового элемента

Выполняет добавление элемента в кучу за время $O(\log n)$. Сначала производится добавление произвольного элемента в конец кучи, затем восстановление свойства упорядоченности с помощью процедуры `siftUp`.

```
function insert(key : int):
    a.heapSize = a.heapSize + 1
    a[a.heapSize - 1] = key
    siftUp(a.heapSize - 1)
```

Реализация на указателях

Для реализации кучи на указателях понадобится представить её в виде обычного бинарного дерева, один элемент которого будет содержать в себе указатель на левого, правого потомков и родителя. Пример на языке Си

```
struct heap_node{
    int key;
    struct heap_node *right, *left, *parent;
};
```

Операции

5 Очереди с приоритетами. Наивная реализация, реализация на бинарной куче

Очередь с приоритетом

Абстрактная структура данных, где у каждого элемента есть приоритет (некоторая величина, приписываемая каждому элементу, помимо значения, которую можно сравнить). Элемент с более высоким приоритетом находится перед элементом с более низким приоритетом. Если у элементов одинаковые приоритеты, они располагаются в зависимости от своей позиции в очереди.

Очередь с приоритетом должна поддерживать стандартные операции:

- `findMin` или `findMax` — поиск элемента с наибольшим приоритетом,
- `insert` или `push` — вставка нового элемента,
- `extractMin` или `extractMax` — извлечь элемент с наибольшим приоритетом,
- `increaseKey` или `decreaseKey` — обновить значение элемента (данной операции, как в стандартной библиотеке C++, может и не быть)

Наивная реализация: (не)упорядоченный список/массив

В одной ячейке которого хранится `key` и `val`, где `key` — приоритет.

5.1 Неотсортированный список

Вставка (`insert`): Сложность: $O(1)$

- Создаётся новый узел, содержащий данные и приоритет.
- Новый узел вставляется в начало списка / конец массива.

Поиск `min/max`: Сложность: $O(n)$

- Проходимся по списку, сравнивая на каждом шаге с текущим и, если больше, то обновляем.
- Сохраняем приоритет на 1 шаге.

Изменение ключа элемента: Сложность: $O(n)$

5.2 Отсортированный массив:

Вставка (insert): Сложность: $O(n)$

- Создаём новый узел.
- Ищем его место в списке/массиве.
- Вставляем.

Поиск max/min: Сложность: $O(1)$

- Первый/последний элемент списка/массива.

Изменение ключа элемента: $O(\log n)$ — например, с помощью бинарного поиска.

5.3 Реализация на бинарной куче

См. билет 6

- **Вставка:** $O(\log n)$
- **Поиск min/max:** $O(1)$ (если куча с min/max в корне)
- **Изменение ключа элемента:** $O(\log n)$

6 Потоки в сетях. Задача о максимальном потоке. Алгоритм Форда – Фалкерсона.

Введём необходимые определения

Сеть $G = (V, E)$ — ориентированный граф, в котором:

1. каждое ребро $(u, v) \in E$ имеет положительную пропускную способность $c(u, v) > 0$ (англ. capacity). Если $(u, v) \notin E$, то $c(u, v) = 0$,
2. выделены две вершины — **исток** s и **сток** t .

Поток (англ. flow) f в G — действительная функция $f : V \times V \rightarrow \mathbb{R}$, удовлетворяющая условиям:

1. $f(u, v) = -f(v, u)$ (антисимметричность);
2. $f(u, v) \leq c(u, v)$ (ограничение пропускной способности), если ребра нет, то $f(u, v) = 0$;
3. $\sum_v f(u, v) = 0$ для всех вершин u , кроме s и t (закон сохранения потока).

Величина потока f определяется как $|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$.

В задаче о **нахождении максимального потока** дана некоторая сеть G с истоком s и стоком t и требуется найти поток максимальной величины.

Алгоритм Форда-Фалкерсона

ddd

dddd

7 Амортизационный анализ: групповой анализ, банковский метод. Амортизационный анализ для бинарного счетчика

В ходе **амортизационного анализа** (amortized analysis) время, необходимое для выполнения последовательности операций над структурой данных, усредняется по всем выполняемым операциям. Этот анализ можно использовать, например, чтобы показать, что, даже если одна из операций последовательности является дорогостоящей, при усреднении по всей последовательности средняя стоимость операций будет небольшой. Амортизационный анализ отличается от анализа средних величин тем, что в нем не учитывается вероятность. При выполнении амортизационного анализа гарантируется *средняя производительность операций в наихудшем случае*.

7.1 Групповой анализ

В ходе **группового анализа** (aggregate analysis) исследователь показывает, что в наихудшем случае суммарное время выполнения последовательности всех n операций равно $T(n)$. Поэтому в наихудшем случае средняя, или амортизиро-анная стоимость (amortized cost), приходящаяся на одну операцию, определяется соотношением $T(n)/n$. Заметим, что такая амортизированная стоимость применима ко всем операциям, даже если в последовательности имеется несколько разных их типов. В других двух методах (методе бухгалтерского учета и методе потенциалов) операциям различного вида могут присваиваться разные амортизированные стоимости.

7.1.1 Увеличение показаний бинарного счетчика

Рассмотрим задачу реализации k -битового бинарного счетчика, который ведет счет от нуля в восходящем направлении. В качестве счетчика используется битовый массив $A[0..k-1]$, где $A.length = k$. Младший бит хранящегося в счетчике бинарного числа x находится в элементе $[0]$, а старший бит — в элементе $A[k-1]$, так что $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Чтобы увеличить показания счетчика на 1 (по модулю 2^k), используется следующая процедура.

Increment(A)

$i = 0$

 while $i < A.length$ и $A[i] == 1$

$A[i] = 0$


```

    i = i + 1
if i < A.length
    A[i] = 1

```

В начале каждой итерации цикла `while` в строках 2–4 мы добавляем 1 к биту в позиции i . Если $A[i] = 1$, то добавление 1 обнуляет бит, который находится на позиции i , и приводит к тому, что добавление 1 будет выполнено и в позиции $i + 1$ на следующей операции цикла. В противном случае цикл оканчивается, так что если по его окончании $i < k$, то $A[i] = 0$ и нам нужно изменить значение i -го бита на 1, что и делается в строке 6. Стоимость каждой операции `Increment` линейно зависит от количества изменённых битов.

Поверхностный анализ даст правильную, но неточную оценку. В наихудшем случае, когда массив A состоит только из единиц, для выполнения операции `Increment` потребуется время $\Theta(k)$. Таким образом, выполнение последовательности из n операций `Increment` для изначально обнуленного счетчика в наихудшем случае займет время $O(nk)$.

Этот анализ можно уточнить, в результате чего для последовательности из n операций `Increment` в наихудшем случае получается стоимость $O(n)$. Такая оценка возможна благодаря тому, что далеко не при каждом вызове процедуры `Increment` изменяются значения всех битов. Например, элемент $A[0]$ изменяется при каждом вызове операции `Increment`. Следующий по старшинству бит $A[1]$ изменяется только через раз, так что последовательность из n операций `Increment` над изначально обнуленным счетчиком приводит к изменению элемента $A[1]$ $\lfloor n/2 \rfloor$ раз. Аналогично бит $A[2]$ изменяется только каждый четвертый раз, т.е. $\lfloor n/4 \rfloor$ раз в последовательности из n операций `Increment` над изначально обнуленным счетчиком. В общем случае для $i = 0, 1, \dots, k - 1$ бит $A[i]$ изменяется $\lfloor n/2^i \rfloor$ раз в последовательности из n операций `Increment` над изначально обнуленным счетчиком. Биты же в позициях $i \geq k$ не изменяются. Таким образом, общее количество изменений битов при выполнении последовательности операций равно

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

Поэтому время выполнения последовательности из n операций `Increment` над изначально обнуленным счетчиком в наихудшем случае равно $O(n)$. Средняя стоимость каждой операции, а следовательно, и амортизированная стоимость операции, равна $O(n)/n = O(1)$.

7.2 Метод бухгалтерского учёта

В методе бухгалтерского учёта (accounting method), применяемом в ходе группового анализа, разные операции оцениваются по-разному, в зависимости от их фактической стоимости. Величина, которая начисляется на операцию, называется амортизированной стоимостью (amortized cost). Если амортизированная стоимость операции превышает ее фактическую стоимость, то соответствующая разность присваивается определенным объектам структуры данных как кредит (credit). Кредит можно использовать впоследствии для компенсирующих выплат на операции, амортизированная стоимость которых меньше их фактической стоимости. Таким образом, можно полагать, что амортизированная стоимость операции состоит из ее фактической стоимости и кредита, который либо накапливается, либо расходуется. Этот метод существенно отличается от группового анализа, в котором все операции характеризуются одинаковой амортизированной стоимостью.

К выбору амортизированной стоимости следует подходить с осторожностью. Если нужно провести анализ с использованием амортизированной стоимости, чтобы показать, что в наихудшем случае средняя стоимость операции невелика, полная амортизированная стоимость последовательности операций должна быть верхней границей полной фактической стоимости последовательности. Более того, как и в групповом анализе, это соотношение должно соблюдаться для всех последовательностей операций. Если обозначить фактическую стоимость i -й операции через c_i , а амортизированную стоимость i -й операции через \hat{c}_i , то указанное требование для всех последовательностей, состоящих из n операций, можно выразить следующим образом:

$$\sum_{i=0}^n \hat{c}_i \geq \sum_{i=0}^n c_i.$$

Общий кредит, хранящийся в структуре данных, представляет собой разность между полной амортизированной стоимостью и полной фактической стоимостью, или $\sum_{i=0}^n \hat{c}_i - \sum_{i=0}^n c_i$. Соответственно, полный кредит, связанный со структурой данных, все время должен быть неотрицательным. Если бы полный кредит в каком-либо случае мог стать отрицательным (в результате недооценки ранних операций с надеждой восполнить счет впоследствии), то полная амортизированная стоимость в тот момент была бы ниже соответствующей фактической стоимости; значит, для последовательности операций полная амортизированная стоимость не была бы в этот момент времени верхней границей полной фактической стоимости. Таким образом, необходимо позаботиться о том, чтобы полный кредит для структуры данных никогда не становился отрицательным.

7.2.1 Увеличение показаний бинарного счетчика

В качестве примера, иллюстрирующего метод бухгалтерского учета, проанализируем операцию Increment, которая выполняется над бинарным изначально обнуленным счетчиком. Ранее мы убедились, что время выполнения этой операции пропорционально количеству битов, изменяющих свое значение. В данном примере это количество используется в качестве стоимости. Для представления каждой единицы затрат (в данном случае — изменения битов) будет использован денежный счет.

Чтобы провести амортизационный анализ, начислим на операцию, при которой биту присваивается значение 1 (т.е. бит устанавливается), амортизированную стоимость, равную 2 долларам. Когда бит устанавливается, 1 доллар (из двух начисленных) расходуется на оплату операции по самой установке. Оставшийся 1 доллар вкладывается в этот бит в качестве кредита для последующего использования при его обнулении. В любой момент времени с каждой единицей содержащегося в счетчике значения связан 1 доллар кредита, поэтому для обнуления бита нет необходимости начислять какую-либо сумму; за сброс бита достаточно будет уплатить 1 доллар.

Теперь можно определить амортизированную стоимость операции Increment. Стоимость обнуления битов в цикле while выплачивается за счет тех денег, которые связаны с этими битами. В процедуре Increment устанавливается не более одного бита (в строке 6), поэтому амортизированная стоимость операции Increment не превышает 2 долларов. Количество единиц в бинарном числе, представляющем показания счетчика, не может быть отрицательным, поэтому и сумма кредита всегда неотрицательна. Таким образом, полная амортизированная стоимость n операций Increment равна $O(n)$. Это и есть оценка полной фактической стоимости.

8 Амортизационный анализ: метод потенциалов для динамического массива.

Вместо представления предоплаченной работы в виде кредита, хранящегося в структуре данных вместе с отдельными объектами, в ходе амортизированного анализа по **методу потенциалов** (potential method) такая работа представляется в виде “потенциальной энергии”, или просто “потенциала”, который можно высвободить для оплаты последующих операций. Этот потенциал связан со структурой данных в целом, а не с её отдельными объектами.

Метод потенциалов работает следующим образом. Мы начинаем с исходной структуры данных D_0 , над которой выполняется n операций. Для всех $i = 1, 2, \dots, n$ обозначим через c_i фактическую стоимость i -й операции, а через D_i — структуру данных, которая получается в результате применения i -й операции к структуре данных D_{i-1} . **Функция потенциала** (potential function) Φ отображает каждую структуру данных D_i на действительное число $\Phi(D_i)$, которое является **потенциалом** (potential), связанным со структурой данных D_i . **Амортизированная стоимость** (amortized cost) \hat{c}_i i -й операции определяется соотношением

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Таким образом, амортизированная стоимость каждой операции представляет собой ее фактическую стоимость плюс приращение потенциала в результате выполнения операции. Тогда полная амортизированная стоимость n операций равна

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0).$$

Если функцию потенциала Φ можно определить таким образом, чтобы выполнялось неравенство $\Phi(D_n) \geq \Phi(D_0)$, то полная амортизированная стоимость $\sum_{i=1}^n \hat{c}_i$ является верхней границей полной фактической стоимости $\sum_{i=1}^n c_i$. На практике не всегда известно, сколько операций может быть выполнено, поэтому, если наложить условие $\Phi(D_i) \geq \Phi(D_0)$ для всех i , то, как и в методе бухгалтерского учета, будет обеспечена предоплата. Часто удобно определить величину $\Phi(D_0)$ равной нулю, а затем показать, что для всех i выполняется неравенство $\Phi(D_i) \geq 0$.

8.0.1 Расширение динамической таблицы

В некоторых приложениях заранее не известно, сколько элементов будет храниться в таблице. Может возникнуть ситуация, когда для таблицы

выделяется место, а впоследствии оказывается, что его недостаточно. В этом случае приходится выделять больший объем памяти и копировать все объекты из исходной таблицы в новую, большего размера. Аналогично, если из таблицы удаляется много объектов, может понадобиться преобразовать ее в таблицу меньшего размера. Рассмотрим задачу о динамическом расширении и сжатии таблицы. Методом амортизационного анализа будет показано, что амортизированная стоимость вставки и удаления равна всего лишь $O(1)$, даже если фактическая стоимость операции больше из-за того, что она приводит к расширению или сжатию таблицы.

Предполагается, что в динамической таблице поддерживаются операции Table-Insert и Table-Delete. В результате выполнения операции Table-Insert в таблицу добавляется элемент, занимающий одну ячейку (slot), — пространство для одного элемента. Аналогично операцию Table-Delete можно представлять как удаление элемента из таблицы, в результате чего одна ячейка освобождается.

Предположим, что место для хранения таблицы выделяется в виде массива ячеек. В некоторых программных средах при попытке вставить элемент в заполненную таблицу не остается ничего другого, как прибегнуть к аварийному завершению программы, сопровождаемому выдачей сообщения об ошибке. Однако мы предполагаем, что наша программная среда, подобно многим современным средам, обладает системой управления памятью, позволяющей по запросу выделять и освобождать блоки памяти. Таким образом, когда в заполненную таблицу вставляется элемент, ее можно **расширить** (expand), выделив место для новой таблицы, содержащей больше ячеек, чем было в старой. Поскольку таблица всегда должна размещаться в непрерывной области памяти, для большей таблицы необходимо выделить новый массив, а затем скопировать элементы из старой таблицы в новую.

Общепринятый эвристический подход заключается в том, чтобы в новой таблице было в два раза больше ячеек, чем в старой. Если в таблицу элементы только вставляются, то значение ее коэффициента заполнения будет не меньше $1/2$, поэтому объем неиспользованного места никогда не превысит половины полного размера таблицы.

В приведенном ниже псевдокоде предполагается, что T — объект, представляющий таблицу. Атрибут $T.table$ содержит указатель на блок памяти, представляющий таблицу. В $T.num$ содержится количество элементов в таблице, а в $T.size$ — полное количество ячеек в таблице. Изначально таблица пустая: $T.num = T.size = 0$.

```
Table-Insert (T, x)
    if T.size == 0
```

```

    Выделить  $T.table$  с 1 ячейкой
     $T.size = 1$ 
if  $T.num == T.size$ 
    Выделить new-table с 2  $T.size$  ячейками
    Вставить все элементы из  $T.table$  в new-table
    Освободить  $T.table$ 
     $T.table = new-table$ 
     $T.size = 2 T.size$ 
Вставить  $x$  в  $T.table$ 
 $T.num = T.num + 1$ 

```

Здесь имеется две “вставки” (сама процедура Table-Insert и операция **элементарной вставки** (elementary insertion) в таблицу), выполняемые в строках 6 и 10. Время работы процедуры Table-Insert можно проанализировать в терминах количества элементарных вставок, считая стоимость каждой такой операции равной 1. Предполагается, что фактическое время работы процедуры Table-Insert линейно зависит от времени вставки отдельных элементов, так что накладные расходы на выделение исходной таблицы в строке 2 — константа, а накладные расходы на выделение и освобождение памяти в строках 5 и 7 пренебрежимо малы по сравнению со стоимостью переноса элементов в строке 6. Назовем **расширением** (expansion) событие, при котором выполняются строки 5–9.

Определим функцию потенциала Φ , которая становится равной 0 сразу после расширения и достигает значения, равного размеру матрицы, к тому времени, когда матрица станет заполненной. В этом случае предстоящее расширение можно будет оплатить за счет потенциала. Одним из возможных вариантов является функция

$$\Phi(T) = 2 \cdot T.num - T.size.$$

Сразу после расширения выполняется соотношение $T.num = T.size/2$, поэтому, как и требуется, $\Phi(T) = 0$. Непосредственно перед расширением справедливо равенство $T.num = T.size$, следовательно, как и требуется, $\Phi(T) = T.num$. Начальное значение потенциала равно 0, и, поскольку таблица всегда заполнена не менее чем наполовину, выполняется неравенство $T.num \geq T.size/2$, из которого следует, что функция $\Phi(T)$ всегда неотрицательна. Таким образом, суммарная амортизированная стоимость n операций Table-Insert является верхней границей суммарной фактической стоимости.

Чтобы проанализировать амортизированную стоимость i -й операции Table-Insert, обозначим через num_i количество элементов, хранящихся в таблице после этой операции, через $size_i$ — общий размер таблицы после

этой операции и через Φ_i — потенциал после этой операции. Изначально $num_0 = 0$, $size_0 = 0$ и $\Phi_0 = 0$. Если i -я операция Table-Insert не приводит к расширению, то $size_i = size_{i-1}$ и амортизированная стоимость операции равна

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + (2 \cdot num_i - size_i) - (2 \cdot (num_i - 1) - size_i) \\
&= 3.
\end{aligned}$$

Если же i -я операция Table-Insert приводит к расширению, то $size_i = 2 \cdot size_{i-1}$ и $size_{i-1} = num_{i-1} = num_i - 1$, откуда вытекает, что $size_i = 2 \cdot (num_i - 1)$. Таким образом, амортизированная стоимость операции равна

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= num_i + (2 \cdot num_i - (num_i - 1)) - (2 \cdot (num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3.
\end{aligned}$$

9 Алгоритм Рабина — Карпа, алгоритм Кнута — Морриса — Пратта. Структура данных «бор», алгоритм Ахо — Корасик

9.1 Алгоритм Рабина — Карпа

Алгоритм Рабина — Карпа предназначен для поиска подстроки в строке. Наивный алгоритм поиска подстроки в строке работает за $O(n^2)$ в худшем случае — слишком долго. Чтобы ускорить этот процесс, можно воспользоваться методом хеширования.

Опр. 9.1. Пусть дана строка $s[0..n-1]$. Тогда **полиномиальным хешем** (англ. polynomial hash) строки s называется число $h = hash(s[0..n-1]) = p^0s[0] + \dots + p^{n-1}s[n-1]$, где p — некоторое простое число, а $s[i]$ — код i -го символа строки s .

Проблему переполнения при вычислении хешей довольно больших строк можно решить так — считать хеши по модулю $r = 2^{64}$ (или 2^{32}), чтобы модуль брался автоматически при переполнении типов. Для работы алгоритма потребуется считать хеш подстроки $s[i..j]$. Делать это можно следующим образом:

Рассмотрим хеш $s[0..j]$:

$$hash(s[0..j]) = s[0] + ps[1] + \dots + p^{i-1}s[i-1] + p^is[i] + \dots + p^{j-1}s[j-1] + p^js[j].$$

Разобьем это выражение на две части:

$$hash(s[0..j]) = (s[0] + ps[1] + \dots + p^{i-1}s[i-1]) + (p^is[i] + \dots + p^{j-1}s[j-1] + p^js[j]).$$

Вынесем из последней скобки множитель p^i :

$$hash(s[0..j]) = (s[0] + ps[1] + \dots + p^{i-1}s[i-1]) + p^i(s[i] + \dots + p^{j-i-1}s[j-1] + p^{j-i}s[j]).$$

Выражение в первой скобке есть не что иное, как хеш подстроки $s[0..i-1]$, а во второй — хеш нужной нам подстроки $s[i..j]$. Итак, мы получили, что:

$$hash(s[0..j]) = hash(s[0..i-1]) + p^i hash(s[i..j]).$$

Отсюда получается следующая формула для $hash(s[i..j])$:

$$hash(s[i..j]) = (1/p^i)(hash(s[0..j]) - hash(s[0..i-1])).$$

Однако, как видно из формулы, чтобы уметь считать хеш для всех подстрок, начинающихся с i , нужно предсчитать все p^i для $i \in [0..n-1]$. Это займет много памяти. Но поскольку нам нужны только подстроки

размером m , мы можем подсчитать хеш подстроки $s[0..m - 1]$, а затем пересчитывать хеши для всех $i \in [0..n - m]$ за $O(1)$ следующим образом:

$$\text{hash}(s[i + 1..i + m - 1]) = (\text{hash}(s[i..i + m - 1]) - p^{m-1}s[i]) \mod r.$$

$$\text{hash}(s[i + 1..i + m]) = (p \cdot \text{hash}(s[i + 1..i + m - 1]) + s[i + m]) \mod r.$$

Получается:

$$\text{hash}(s[i + 1..i + m]) = (p \cdot \text{hash}(s[i..i + m - 1]) - p^i s[i] + s[i + m]) \mod r.$$

9.1.1 Алгоритм

Алгоритм начинается с подсчета $\text{hash}(s[0..m - 1])$ и $\text{hash}(p[0..m - 1])$, а также с подсчета p^m , для ускорения ответов на запрос.

Для $i \in [0..n - m]$ вычисляется $\text{hash}(s[i..i + m - 1])$ и сравнивается с $\text{hash}(p[0..m - 1])$. Если они оказались равны, то образец p , скорее всего, содержится в строке s начиная с позиции i , хотя возможны и ложные срабатывания алгоритма. Если требуется свести такие срабатывания к минимуму или исключить вовсе, то применяют сравнение некоторых символов из этих строк, которые выбраны случайным образом, или применяют явное сравнение строк, как в наивном алгоритме поиска подстроки в строке. В первом случае проверка произойдет быстрее, но вероятность ложного срабатывания, хоть и небольшая, останется. Во втором случае проверка займет время, равное длине образца, но полностью исключит возможность ложного срабатывания.

Если требуется найти индексы вхождения нескольких образцов, или сравнить две строки — выгоднее будет предпосчитать все степени p , а также хеши всех префиксов строки s .

9.1.2 Псевдокод

Приведем пример псевдокода, который находит все вхождения строки w в строку s и возвращает массив позиций, откуда начинаются вхождения.

```
vector<int> rabinKarp (s : string, w : string):
    vector<int> answer
    int n = s.length
    int m = w.length
    int hashS = hash(s[0..m - 1])
    int hashW = hash(w[0..m - 1])
    for i = 0 to n - m
        if hashS == hashW
            answer.add(i)
        hashS = (p * hashS - pm
```

```
* hash(s[i]) + hash(s[i + m])) mod r // r - некоторое большое число, p -
return answer
```

Новый хеш $hashS$ был получен с помощью быстрого пересчёта. Для сохранения корректности алгоритма нужно считать, что $s[n + 1]$ — пустой символ.

9.1.3 Время работы

Изначальный подсчёт хешей выполняется за $O(m)$. Каждая итерация выполняется за $O(1)$. В цикле всего $n - m + 1$ итераций. Итоговое время работы алгоритма $O(n + m)$.

Однако если требуется исключить ложные срабатывания алгоритма полностью, т.е. придется проверить все полученные позиции вхождения на истинность, то в худшем случае итоговое время работы алгоритма будет $O(n \cdot m)$.

9.2 Алгоритм Кнута — Морриса — Пратта

Алгоритм Кнута — Морриса — Пратта (англ. Knuth–Morris–Pratt algorithm) — алгоритм поиска подстроки в строке.

9.2.1 Алгоритм

Дана цепочка T и образец P . Требуется найти все позиции, начиная с которых P входит в T . Построим строку $S = P\#T$, где $\#$ — любой символ, не входящий в алфавит P и T . Посчитаем на ней значение префикс-функции p . Благодаря разделительному символу $\#$ выполняется $\forall i : p[i] \leq |P|$. Заметим, что по определению префикс-функции при $i > |P|$ и $p[i] = |P|$ подстроки длины P , начинающиеся с позиций 0 и $i - |P| + 1$, совпадают. Соберем все такие позиции $i - |P| + 1$ строки S , вычтем из каждой позиции $|P| + 1$, это и будет ответ. Другими словами, если в какой-то позиции i выполняется условие $p[i] = |P|$, то в этой позиции начинается очередное вхождение образца в цепочку.

9.2.2 Псевдокод

```
int[] kmp(string P, string T):
    int pl = P.length
    int tl = T.length
    int[] answer
    int[] p = prefixFunction(P + "#" + T)
    int count = 0
```

```

for i = 0 .. t1 - 1
    if p[pl + i + 1] == pl
        answer[count++] = i - pl
return answer

```

9.2.3 Время работы

Префикс-функция от строки S строится за $O(S) = O(P + T)$. Проход цикла по строке S содержит $O(T)$ итераций. Суммарное время работы алгоритма оценивается как $O(P + T)$.

9.2.4 Оценка по памяти

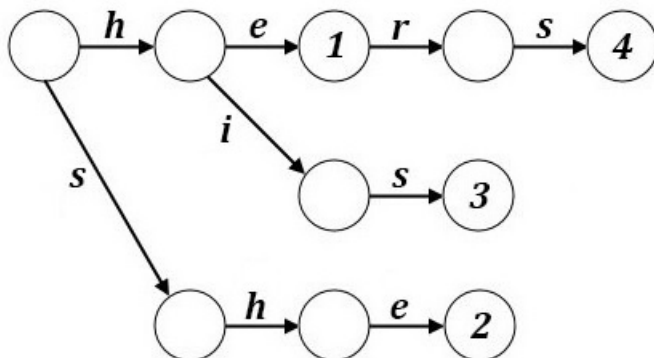
Предложенная реализация имеет оценку по памяти $O(P + T)$. Оценки $O(P)$ можно добиться за счет запоминания значений префикс-функции для позиций в S , меньших $|P| + 1$ (то есть до начала цепочки T). Это возможно, так как значение префикс-функции не может превысить длину образца благодаря разделительному символу $\#$.

9.3 Структура данных «бор»

Опр. 9.2. Бор (англ. trie, луч, нагруженное дерево) — структура данных для хранения набора строк, представляющая из себя дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной. Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца.

9.3.1 Пример

Бор для набора образцов {he, she, his, hers}:



9.3.2 Построение

Введем следующие обозначения:

1. S — используемый алфавит;
2. $P = P_1, \dots, P_k$ — набор строк над S , называемый словарём;
3. $n = \sum_{i=1}^k |P_i|$ — сумма длин строк.

Бор храним как набор вершин, у каждой из которых есть метка, обозначающая, является ли вершина терминальной и указатели (рёбра) на другие вершины или на NULL.

```
struct vertex:
    vertex next[|S|]
    bool isTerminal
```

1. Создадим дерево из одной вершины (в нашем случае — корня).
2. Добавление элементов в дерево. Добавляем шаблоны P_i один за другим. Следуем из корня по рёбрам, отмеченным буквами из P_i , пока возможно. Если P_i заканчивается в v , сохраняем идентификатор P_i (например, i) в v и отмечаем вершину v как терминальную. Если ребра, отмеченного очередной буквой P_i , нет, то создаем новое ребро и вершину для символа строки P_i .

Построение занимает, очевидно, $O(|P_1| + \dots + |P_k|) = O(n)$ времени, так как поиск буквы, по которой нужно переходить, происходит за $O(1)$. Поскольку на каждую вершину приходится $O(|S|)$ памяти, то использование памяти есть $O(n|S|)$.

9.4 Алгоритм Ахо — Корасик.

Пусть дан набор строк в алфавите размера k суммарной длины m . Необходимо найти для каждой строки все ее вхождения в текст.

9.4.1 Шаг 1. Построение бора

Строим бор из строк. Построение выполняется за время $O(m)$, где m — суммарная длина строк.

9.4.2 Шаг 2. Преобразование бора

Обозначим за $[u]$ слово, приводящее в вершину u в боре. Узлы бора можно понимать как состояния автомата, а корень как начальное состояние. Узлы бора, в которых заканчиваются строки, становятся терминальными. Для переходов по автомату наведём в узлах несколько функций:

1. $parent(u)$ — возвращает родителя вершины u ;

2. $\pi(u) = \delta(\pi(\text{parent}(u)), c)$ — **суффиксная ссылка**, и существует переход из $\text{parent}(u)$ в u по символу c ;

3. Функция перехода $\delta(u, c) = \begin{cases} v, & \text{если из } u \text{ в } v \text{ ведёт символ } c; \\ \text{root}, & \text{если } u \text{ — корень и из него не исходит символ } c; \\ \delta(\pi(u), c), & \text{иначе.} \end{cases}$

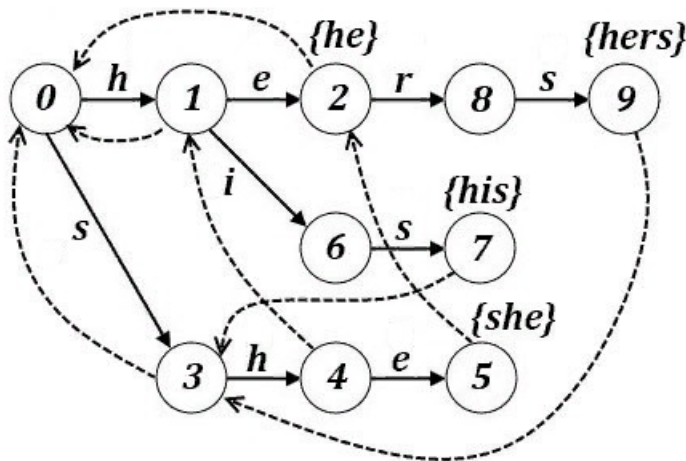
Мы можем понимать рёбра бора как переходы в автомате по соответствующей букве. Однако одними только рёбрами бора нельзя ограничиваться. Если мы пытаемся выполнить переход по какой-либо букве, а соответствующего ребра в боре нет, то мы тем не менее должны перейти в какое-то состояние. Для этого нам и нужны суффиксные ссылки. Суффиксная ссылка $\pi(u) = v$, если $[v]$ — максимальный суффикс $[u]$, $[v] \neq [u]$. Функции перехода и суффиксные ссылки можно найти либо алгоритмом обхода в глубину с ленивыми вычислениями, либо с помощью алгоритма обхода в ширину.

Из определений выше можно заметить два следующих факта:

- функция перехода определена через суффиксную ссылку, а суффиксная ссылка — через функцию переходов;
- для построения суффиксных ссылок необходимо знать информацию только выше по бору от текущей вершины до корня.

Это позволяет реализовать функции поиска переходов по символу и суффиксных ссылок ленивым образом при помощи взаимной рекурсии.

9.4.3 Пример автомата Ахо-Корасик



Пунктиром обозначены суффиксные ссылки. Из вершин, для которых они не показаны, суффиксные ссылки идут в корень.

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.

9.4.4 Шаг 3. Построение сжатых суффиксных ссылок

При построении автомата может возникнуть такая ситуация, что ветвление есть не на каждом символе. Тогда можно маленький бамбук заменить одним ребром. Для этого и используются сжатые суффиксные ссылки.

$$ur(u) = \begin{cases} \pi(u), & \text{если } \pi(u) \text{ — терминальная;} \\ \emptyset, & \text{если } \pi(u) \text{ — корень;} \\ ur(\pi(u)), & \text{иначе.} \end{cases}$$

Здесь ur — сжатая суффиксная ссылка, т.е. ближайшее допускающее состояние (терминал) перехода по суффиксным ссылкам. Аналогично обычным суффиксным ссылкам, сжатые суффиксные ссылки могут быть найдены при помощи ленивой рекурсии.

9.4.5 Использование автомата

По очереди просматриваем символы текста. Для очередного символа s переходим из текущего состояния u в состояние, которое вернёт функция $\delta(u, s)$. Оказавшись в новом состоянии, отмечаем по сжатым суффиксным ссылкам строки, которые нам встретились, и их позицию (если требуется). Если новое состояние является терминальным, то соответствующие ему строки тоже отмечаем.

10 Алгоритм Бойера-Мура. Эвристики стоп-символа и хорошего суффикса.

Цель: найти все вхождения шаблона в строку, используя при этом как можно меньше дополнительной памяти (в отличие от алгоритмов Кнута — Морриса — Пратта и Ахо — Корасика)

Идея: возьмём базовый алгоритм для поиска за $O(nk)$

- Проходимся по всем `str1` от i -го до $i + \text{len}(\text{str2}) - 1$,
- Посимвольно сравниваем,

и попробуем его модернизировать

Сам алгоритм: Будем сравнивать все символы в строке с символами подстроки **справа налево**. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и поиск окончен. В случае несовпадения какого-либо символа (или полного совпадения всего шаблона) он использует две предварительно вычисляемых эвристических функций, чтобы сдвинуть позицию для начала сравнения вправо.

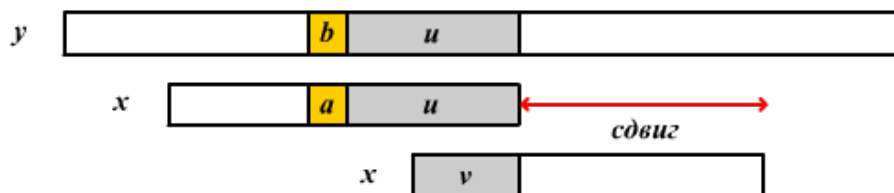
Будем применять **две эвристики**:

- Эвристика хорошего суффикса.
- Эвристика стоп-символа.

Эвристика хорошего суффикса

Если при сравнении текста и шаблона совпало один или больше символов, шаблон сдвигается в зависимости от того, какой суффикс совпал.

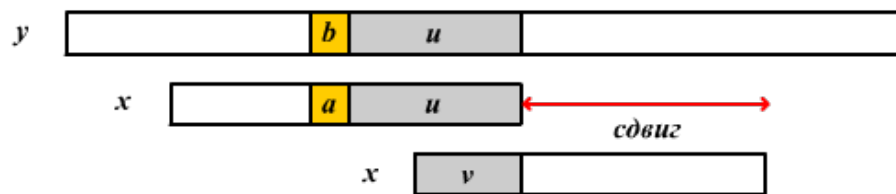
Если встретились `oo`



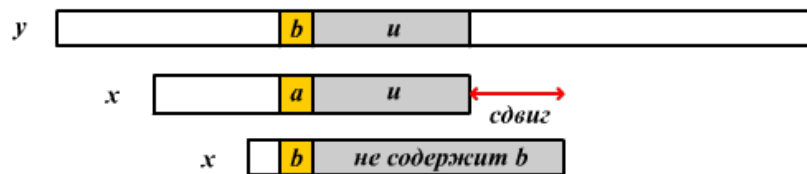
Сдвиг хорошего суффикса, вся подстрока `u` полностью встречается справа от символа `s`, отличного от символа `a`

10.1 Эвристика стоп-символа

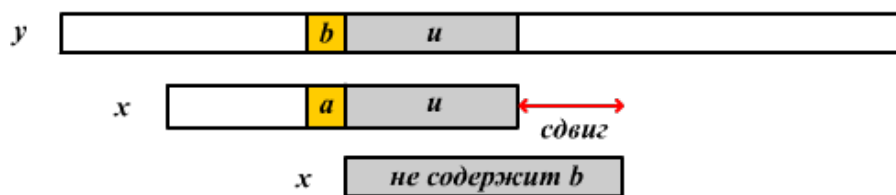
На этапе инициализации составляется таблица плохих символов, в которой у каждого символа из алфавита отмечается последняя позиция его вхождения в шаблон.



Сдвиг хорошего суффикса, только суффикс подстроки u повторно встречается в x .



Сдвиг плохого символа, символ a входит в x .



Сдвиг плохого символа, символ b не входит в x .

Псевдокод алгоритма

Константой $|\Sigma| = \sigma$ обозначим размер нашего алфавита.

Функция для вычисления таблицы плохих символов

11 Расстояние Левенштейна, алгоритм Вагнера — Фишера

Опр. 11.1. Расстояние Левенштейна (англ. Levenshtein distance) (также редакционное расстояние или дистанция редактирования) между двумя строками в теории информации и компьютерной лингвистике — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Для расстояния Левенштейна справедливы следующие утверждения:

- $d(S_1, S_2) \geq ||S_1| - |S_2||$,
- $d(S_1, S_2) \leq \max(|S_1|, |S_2|)$,
- $d(S_1, S_2) = 0 \Leftrightarrow S_1 = S_2$,

где $d(S_1, S_2)$ — расстояние Левенштейна между строками S_1 и S_2 , а $|S|$ — длина строки S . Расстояние Левенштейна является метрикой. Для того, чтобы доказать это, достаточно доказать, что выполняется неравенство треугольника:

- $d(S_1, S_3) \leq d(S_1, S_2) + d(S_2, S_3)$.

Пусть $d(S_1, S_3) = x$, $d(S_1, S_2) = y$, $d(S_2, S_3) = z$. Тогда x — кратчайшее редакционное расстояние от S_1 до S_3 , y — кратчайшее редакционное расстояние от S_1 до S_2 , а z — кратчайшее редакционное расстояние от S_2 до S_3 . При этом $y + z$ — какое-то расстояние от S_1 до S_3 . В других случаях $d(S_1, S_3) < d(S_1, S_2) + d(S_2, S_3)$. Следовательно, выполняется неравенство треугольника.

Рассмотрим более общий случай: пусть цены операций могут зависеть от вида операции (вставка, удаление, замена) и/или от участвующих в ней символов.

Лемма 11.1. Будем считать, что элементы строк нумеруются с первого, как принято в математике, а не нулевого. Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда редакционное расстояние $d(S_1, S_2)$ можно подсчитать по следующей рекуррентной формуле:

$d(S_1, S_2) = D(M, N)$, где

$$D(i, j) = \begin{cases} 0 & ; i = 0, j = 0 \\ i \cdot deleteCost & ; j = 0, i > 0 \\ j \cdot insertCost & ; i = 0, j > 0 \\ D(i - 1, j - 1) & ; S_1[i] = S_2[j] \\ \min(D(i, j - 1) + insertCost, & \\ \quad D(i - 1, j) + deleteCost, & ; i > 0, j > 0, S_1[i] \neq S_2[j] \\ \quad D(i - 1, j - 1) + replaceCost). & \end{cases}$$

Доказательство. Рассмотрим формулу более подробно. Здесь $D(i, j)$ — расстояние между префиксами строк: первыми i символами строки S_1 и первыми j символами строки S_2 . Очевидно, что редакционное расстояние между двумя пустыми строками равно нулю. Также очевидно то, что чтобы получить пустую строку из строки длиной i , нужно совершить i операций удаления, а чтобы получить строку длиной j из пустой, нужно произвести j операций вставки. Осталось рассмотреть нетривиальный случай, когда обе строки непусты.

Для начала заметим, что в оптимальной последовательности операций их можно произвольно менять местами. В самом деле, рассмотрим две последовательные операции:

- Две замены одного и того же символа — неоптимально (если мы заменили x на y , потом y на z , выгоднее было сразу заменить x на z).
- Две замены разных символов можно менять местами.
- Два стирания или две вставки можно менять местами.
- Вставка символа с его последующим стиранием — неоптимально (можно их обе отменить).
- Стирание и вставку разных символов можно менять местами.
- Вставка символа с его последующей заменой — неоптимально (излишняя замена).
- Вставка символа и замена другого символа меняются местами.
- Замена символа с его последующим стиранием — неоптимально (излишняя замена).
- Стирание символа и замена другого символа меняются местами.

Пусть S_1 кончается на символ a , S_2 кончается на символ b . Есть три варианта:

1. Символ a , на который кончается S_1 , в какой-то момент был стёрт. Сделаем это стирание первой операцией. Тогда мы стёрли символ a , после чего превратили первые $i - 1$ символов S_1 в S_2 (на что потребовалось $D(i - 1, j)$ операций), значит, всего потребовалось $D(i - 1, j) + 1$ операций
2. Символ b , на который кончается S_2 , в какой-то момент был добавлен. Сделаем это добавление последней операцией. Мы превратили S_1 в первые $j - 1$ символов S_2 , после чего добавили b . Аналогично предыдущему случаю, потребовалось $D(i, j - 1) + 1$ операций.
3. Оба предыдущих утверждения неверны. Если мы добавляли символы справа от финального a , то чтобы сделать последним символом b , мы должны были или в какой-то момент добавить его (но тогда утверждение 2 было бы верно), либо заменить на него один из этих добавленных символов (что тоже невозможно, потому что добавление символа с его последующей заменой неоптимально). Значит, символов справа от финального a мы не добавляли. Самого финального a мы не стирали, поскольку утверждение 1 неверно. Значит, единственный способ изменения последнего символа — его замена. Заменять его 2 или больше раз неоптимально. Значит,
 - (a) Если $a = b$, мы последний символ не меняли. Поскольку мы его также не стирали и не приписывали ничего справа от него, он не влиял на наши действия, и, значит, мы выполнили $D(i - 1, j - 1)$ операций.
 - (b) Если $a \neq b$, мы последний символ меняли один раз. Сделаем эту замену первой. В дальнейшем, аналогично предыдущему случаю, мы должны выполнить $D(i - 1, j - 1)$ операций, значит, всего потребуется $D(i - 1, j - 1) + 1$ операций.

11.1 Алгоритм Вагнера — Фишера

Для нахождения кратчайшего расстояния необходимо вычислить матрицу D , используя вышеприведённую формулу. Её можно вычислять как по строкам, так и по столбцам. Псевдокод алгоритма, написанный при произвольных ценах замен, вставок и удалений (важно помнить, что элементы нумеруются с 1):

```
int levensteinInstruction(String s1, String s2,  
                           int InsertCost, int DeleteCost, int ReplaceCost):
```

```

D[0][0] = 0
for j = 1 to N
    D[0][j] = D[0][j - 1] + InsertCost
for i = 1 to M
    D[i][0] = D[i - 1][0] + DeleteCost
    for j = 1 to N
        if S1[i] != S2[j]
            D[i][j] = min(D[i - 1][j] + DeleteCost,
                           D[i][j - 1] + InsertCost,
                           D[i - 1][j - 1] + ReplaceCost)
        else
            D[i][j] = D[i - 1][j - 1]
return D[M][N]

```

Этот псевдокод решает простой частный случай, когда вставка символа, удаление символа и замена одного символа на другой стоят одинаково для любых символов.

Алгоритм в виде, описанном выше, требует $\Theta(M \cdot N)$ операций и такую же память, однако, если требуется только расстояние, легко уменьшить требуемую память до $\Theta(N)$. Заметим, что для вычисления $D[i]$ нам нужно только $D[i - 1]$, поэтому будем вычислять $D[i]$ в $D[1]$, а $D[i - 1]$ в $D[0]$. Осталось только поменять местами $D[1]$ и $D[0]$.

```

int levensteinInstruction(int[] D):
    for i = 0 to M
        for j = 0 to N
            вычислить D[1][j]
        swap(D[0], D[1])
    return D[0][N]

```

12 Сканирующая прямая. Алгоритм Бентли-Оттоманна для поиска пересечения отрезков

13 Диаграммы Вороного. Алгоритм Форчуна.

14 Триангуляция Делоне, связь с диаграммами Вороного. Алгоритм построения

15 Сумма Минковского. Задача планирования движения робота в среде с препятствиями. Граф видимости.

16 Отсечение невидимых поверхностей. Z-буфер и алгоритм художника.