1 QuickSort

Быстрый метод сортировки функционирует по принципу "разделяй и властвуй".

1. Массив $a[l \dots r]$ типа T разбивается на два (возможно пустых) подмассива $a[l \dots q]$ и $a[q+1 \dots r]$, таких, что каждый элемент $a[l \dots q]$ меньше или равен $a[q]$, который в свою очередь, не превышает любой элемент подмассива $a[q+1 \dots r]$. Индекс вычисляется в ходе процедуры разбиения.

2. Подмассивы $a[l \dots q]$ и $a[q+1 \dots r]$ сортируются с помощью рекурсивного вызова процедуры быстрой сортировки.

3. Поскольку подмассивы сортируются на месте, для их объединения не требуются никакие действия: весь массив $a[l \dots r]$ оказывается отсортированным.

```
void quicksort(a: T[n], int l, int r)
    if l < r
        int q = partition(a, l, r)
        quicksort(a, l, q)
        quicksort(a, q + 1, r)

int partition(a: T[n], int l, int r)
    T v = a[(l + r) / 2]
    int i = l
    int j = r
    while (i <= j)
        while (a[i] < v)
            i++
        while (a[j] > v)
            j—
        if (i >= j)
            break
        swap(a[i++], a[j−−])
    return j
```

Требуемая память. $O(\log n)$ (стек вызовов).

Худшее время работы. Предположим, что мы разбиваем массив так, что одна часть содержит $n-1$ элементов, а вторая – 1. Поскольку процедура разбиения занимает время $\Theta(n)$, для времени работы $T(n)$ получаем соотношение: $T(n) = T(n-1) + \Theta(n) = \sum\limits_{k=1}^{n} \Theta(k) = \Theta(\sum\limits_{k=1}^{n} k) = \Theta(n^2)$.

Среднее время работы. Время работы алгоритма быстрой сортировки равно $O(n \log n)$.

Пусть X – полное количество сравнений элементов с опорным за время работы сортировки. Переименуем элементы массива как $z_1 \dots z_n$, где z_i наименьший по порядку элемент. Также введем множество $Z_{ij} = \{z_i, z_{i+1} \dots z_j\}$.

Заметим, что сравнение каждой пары элементов происходит не больше одного раза, так как элемент сравнивается с опорным, а опорный элемент после разбиения больше не будет участвовать в сравнении.

Поскольку каждая пара элементов сравнивается не более одного раза, полное количество сравнений выражается как

$X = \sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} X_{ij}$, где $X_{ij} = 1$ если произошло сравнение z_i и z_j и $X_{ij} = 0$, если сравнения не произошло.

$E[X] = E\left[\sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} X_{ij}\right] = \sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} E[X_{ij}] = \sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} Pr\{z_i \text{ сравнивается с } z_j\}$

Осталось вычислить величину $Pr\{z_i \text{ сравнивается с } z_j\}$. Поскольку предполагается, что все элементы в массиве различны, то при выборе x в качестве опорного элемента впоследствии не будут сравниваться никакие z_i и z_j для которых $z_i < x < z_j$. С другой стороны, если z_i выбран в качестве опорного, то он будет сравниваться с каждым элементом Z_{ij} кроме себя самого. Таким образом элементы z_i и z_j сравниваются тогда и только тогда когда первым в множестве Z_{ij} опорным элементом был выбран один из них.

$Pr\{z_i \text{ сравнивается с } z_j\} = Pr\{\text{первым опорным элементом был } z_i \text{ или } z_j\} = Pr\{\text{первым опорным элементом был } z_i\} + Pr\{\text{первым опорным элементом был } z_j\} = \dfrac{1}{j-i+1} + \dfrac{1}{j-i+1} = \dfrac{2}{j-i+1}$

$E[X] = \sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} \dfrac{2}{j-i+1} = \sum\limits_{i=1}^{n-1} \sum\limits_{k=1}^{n-i} \dfrac{2}{k+1} < \sum\limits_{i=1}^{n-1} \sum\limits_{k=1}^{n-i} \dfrac{2}{k} = \sum\limits_{i=1}^{n-1} O(\log n) = O(n \log n)$

2 MergeSort

Алгоритм использует принцип «разделяй и властвуй».

1. Если в рассматриваемом массиве один элемент, то он уже отсортирован — алгоритм завершает работу.

2. Иначе массив разбивается на две части, которые сортируются рекурсивно.

3. После сортировки двух частей массива к ним применяется процедура слияния, которая по двум отсортированным частям получает исходный отсортированный массив.

```
function mergeSortRecursive(a : int[n]; left, right : int):
    if left + 1 >= right
        return
    mid = (left + right) / 2
    mergeSortRecursive(a, left, mid)
    mergeSortRecursive(a, mid, right)
    merge(a, left, mid, right)
```

```
function merge(a : int[n]; left, mid, right : int):
    it1 = 0
    it2 = 0
    result : int[right - left]

    while left + it1 < mid and mid + it2 < right
        if a[left + it1] < a[mid + it2]
            result[it1 + it2] = a[left + it1]
            it1 += 1
        else
            result[it1 + it2] = a[mid + it2]
            it2 += 1

    while left + it1 < mid
        result[it1 + it2] = a[left + it1]
        it1 += 1

    while mid + it2 < right
        result[it1 + it2] = a[mid + it2]
        it2 += 1

    for i = 0 to it1 + it2
        a[left + i] = result[i]
```

Требуемая память. $O(n)$.

Время работы. Чтобы оценить время работы этого алгоритма, составим рекуррентное соотношение. Пускай $T(n)$ – время сортировки массива длины n, тогда для сортировки слиянием справедливо
$T(n) = 2T(\frac{n}{2}) + O(n)$, где $O(n)$ – время, необходимое на то, чтобы слить два массива длины n. Распишем это соотношение:
$T(n) = 2T(\frac{n}{2}) + O(n) = 4T(\frac{n}{4}) + 2O(n) = \cdots = T(1) + \log(n)O(n) = O(n\log(n))$.

3 HeapSort

3.1 Двоичная куча.

Двоичная куча — такое двоичное подвешенное дерево, для которого выполнены следующие три условия:

1. Значение в любой вершине не меньше, чем значения её потомков.
2. На i-ом слое $2i$ вершин, кроме последнего.
3. Последний слой заполнен слева направо.

```
function buldHeap():
    for i = a.heapSize / 2 downto 0
        siftDown(i)

function siftDown(i : int):
    while 2 * i + 1 < a.heapSize
        left = 2 * i + 1
        right = 2 * i + 2
        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
```

3.2 HeapSort

Необходимо отсортировать массив A, размером n. Построим на базе этого массива за $O(n)$ кучу для максимума. Так как максимальный элемент находится в корне, то если поменять его местами с $A[n-1]$, он встанет на своё место. Далее вызовем процедуру siftDown(0), предварительно уменьшив heapSize на 1. Она за $O(\log n)$ просеет $A[0]$ на нужное место и сформирует новую кучу (так как мы уменьшили её размер, то куча располагается с $A[0]$ по $A[n-2]$, а элемент $A[n-1]$ находится на своём месте). Повторим эту процедуру для новой кучи, только корень будем менять местами не с $A[n-1]$, а с $A[n-2]$. Делая аналогичные действия, пока heapSize не станет равен 1, мы будем ставить наибольшее из оставшихся чисел в конец не отсортированной части. Очевидно, что таким образом, мы получим отсортированный массив.

```
fun heapSort(A : list <T>):
    buildHeap(A)
    heapSize = A.size
    for i = 0 to n - 2
        swap(A[0], A[n - 1 - i])
        heapSize--
        siftDown(A, 0, heapSize)
```

Требуемая память. $O(1)$.

Время работы. Операция siftDown работает за $O(\log n)$. Всего цикл выполняется $(n-1)$ раз. Таким образом сложность сортировки кучей является $O(n\log n)$.

4 Хеш-таблица, полиномиальная хэш-функция.

4.1 Полиномиальная хэш-функция.

Используется в алгоритме Рабина-Карпа, предназначенного для поиска подстроки в строке.

Пусть дана строка $s[0..n-1]$. Тогда полиномиальным хешем строки s называется число $h = \text{hash}(s[0..n-1]) = p^0 s[0] + ... + p^{n-1} s[n-1]$, где p — некоторое простое число, а $s[i]$ – код i-ого символа строки s.

Проблему переполнения при вычислении хешей довольно больших строк можно решить так – считать хеши по модулю $r = 2^{64}$, чтобы модуль брался автоматически при переполнении типов.

Для работы алгоритма потребуется считать хеш подстроки $s[i..j]$. Делать это можно следующим образом:

Рассмотрим хеш $s[0..j]$:

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

Разобьем это выражение на две части:

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

Вынесем из последней скобки множитель p^i:

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

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:

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

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

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

Однако, как видно из формулы, чтобы уметь считать хеш для всех подстрок начинающихся с i, нужно предпосчитать 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:

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

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

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

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.

```
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 - p^m * hash(s[i]) + hash(s[i + m])) mod r
    return answer
```

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

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 $O(n \cdot m)$.

4.2 Хеш-таблица.

Хеш-таблица – структура данных, реализующая интерфейс ассоциативного массива. Представляет собой эффективную структуру данных для реализации словарей, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

4.3 Разрешение коллизий с помощью цепочек.

Каждая ячейка i массива H содержит указатель на начало списка всех элементов, хеш-код которых равен i, либо указывает на их отсутствие. Коллизии приводят к тому, что появляются списки размером больше одного элемента.

В зависимости от того нужна ли нам уникальность значений операции вставки у нас будет работать за разное время. Если не важна, то мы используем список, время вставки в который будет в худшем случае равна $O(1)$. Иначе мы проверяем есть ли в списке данный элемент, а потом в случае его отсутствия мы его добавляем. В таком случае вставка элемента в худшем случае будет выполнена за $O(n)$

Время работы поиска в наихудшем случае пропорционально длине списка.

Удаления элемента может быть выполнено за $O(1)$, как и вставка, при использовании двухсвязного списка.

4.4 Линейное разрешение коллизий.

Все элементы хранятся непосредственно в хеш-таблице, без использования связных списков. В отличие от хеширования с цепочками, при использовании этого метода может возникнуть ситуация, когда хеш-таблица окажется полностью заполненной, следовательно, будет невозможно добавлять в неё новые элементы. Так что при возникновении такой ситуации решением может быть динамическое увеличение размера хеш-таблицы, с одновременной её перестройкой.

Добавление элемента в таблицу.

Стратегии поиска:

1. Последовательный поиск. При попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки $i+1, i+2, i+3$ и так далее, пока не найдём свободную ячейку. В неё и запишем элемент.

2. Линейный поиск. Выбираем шаг q. При попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки $i+(1\cdot q), i+(2\cdot q), i+(3\cdot q)$ и так далее, пока не найдём свободную ячейку. В неё и запишем элемент.

3. Квадратичный поиск. Шаг q не фиксирован, а изменяется квадратично: $q = 1, 4, 9, 16....$ Соответственно при попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки $i+1, i+4, i+9$ и так далее, пока не найдём свободную ячейку.

Проверка наличия элемента в таблице. Проверка осуществляется аналогично добавлению: мы проверяем ячейку i и другие, в соответствии с выбранной стратегией, пока не найдём искомый элемент или свободную ячейку.

При поиске элемента может получится так, что мы дойдём до конца таблицы. Обычно поиск продолжается, начиная с другого конца, пока мы не придём в ту ячейку, откуда начинался поиск.

Удаление элемента без пометок. Рассуждение будет описывать случай с линейным поиском хеша. Будем при удалении элемента сдвигать всё последующие на q позиций назад. При этом:

1. если в цепочке встречается элемент с другим хешем, то он должен остаться на своём месте (такая ситуация может возникнуть если оставшаяся часть цепочки была добавлена позже этого элемента)

2. в цепочке не должно оставаться "дырок", тогда любой элемент с данным хешем будет доступен из начала цепи

Учитывая это будем действовать следующим образом: при поиске следующего элемента цепочки будем пропускать все ячейки с другим значением хеша, первый найденный элемент копировать в текущую ячейку, и затем рекурсивно его удалять. Если такой следующей ячейки нет, то текущий элемент можно просто удалить, сторонние цепочки при этом не разрушатся (чего нельзя сказать про случай квадратичного поиска).

4.5 Хеширование кукушки.

Основная идея хеширования кукушки — использование двух хеш-функций вместо одной. Рассмотрим алгоритмы функций $add(x)$, $remove(x)$ и $contains(x)$.

Выберем 2 хэш-функции $h_1(x)$ и $h_2(x)$ универсального семейства хэш-функций.

Add. Добавляет элемент с ключом x в хэш-таблицу.

1. Если одна из ячеек с индексами $h_1(x)$ или $h_2(x)$ свободна, кладем в нее элемент.

2. Иначе произвольно выбираем одну из этих ячеек, запоминаем элемент, который там находится, помещаем туда новый.

3. Смотрим в ячейку, на которую указывает другая хеш-функция от элемента, который запомнили, если она свободна, помещаем его в нее.

4. Иначе запоминаем элемент из этой ячейки, кладем туда старый. Проверяем, не зациклились ли мы.

5. Если не зациклились, то продолжаем данную процедуру поиска свободного места пока не найдем свободное место или зациклимся.

6. Иначе выбираем 2 новые хеш-функции и перехешируем все добавленные элементы.

7. Так же после добавления нужно увеличить размер таблицы в случае если она заполнена.

Remove. Удаляет элемент с ключом x из хэш-таблицы.

1. Смотрим ячейки с индексами $h_1(x)$ и $h_2(x)$.

2. Если в одной из них есть искомый элемент, просто помечаем эту ячейку как свободную.

Contains. Проверяет на наличие элемента x в хэш-таблице.

1. Смотрим ячейки с индексами $h_1(x)$ и $h_2(x)$.

2. Если в одной из них есть искомый элемент, возвращаем true.

3. Иначе возвращаем false.

Время работы алгоритма. Удаление и проверка происходят за $O(1)$ (требуется проверить всего лишь 2 ячейки таблицы), добавление в среднем происходит за $O(1)$.

5 Динамическое программирование: общая идея, линейная динамика, матричная, динамика на отрезках.

5.1 Общая идея.

Динамическое программирование — это когда у нас есть задача, которую непонятно как решать, и мы разбиваем ее на меньшие задачи, которые тоже непонятно как решать.

В процессе составления алгоритмов динамического программирования, требуется следовать последовательности из четырёх действий:

1. Описать структуру оптимального решения.
2. Рекурсивно определить значение оптимального решения.
3. Вычислить значение оптимального решения с помощью метода восходящего анализа.
4. Составить оптимальное решение на основе полученной информации.

5.2 Линейная динамика.

Задача (количество последовательностей без трех единиц подряд). Определите количество последовательностей из нулей и единиц длины N (длина - это общее количество нулей и едииниц), в которых никакие три единицы не стоят рядом.

Решение. Рассмотрим последовательности длины N, на первом месте может стоять 0, таких будет $f(N-1)$, при подстановке 1, второе может быть 0, таких будет $f(N-2)$, и 1,таких $f(N-3)$. Так как трех единиц не может быть, по условию, просто просуммируем полученные значения до нужного N, записывая каждый в массив: $f[n] = f[n-1] + f[n-2] + f[n-3]$; ответ будет находиться в $f[n]$.

5.3 Матричная динамика.

Задача (биномиальные коэффициенты). В прямоугольной таблице $N \times M$ в начале игрок находится в левой верхней клетке. За один ход ему разрешается перемещаться в соседнюю клетку либо вправо, либо вниз (влево и вверх перемещаться запрещено). Посчитайте, сколько есть способов у игрока попасть в правую нижнюю клетку.

Решение. Количество способов попасть в данную клетку равно количество способов попасть в передыдующую сверху плюс в передыдущую слева: $a[i,k] = a[i-1,k] + a[i,k-1]$.

5.4 Динамика на отрезках.

Задача (алгоритм Кока-Янгера-Касами). Дана грамматика $G = (N, \Sigma, P, S)$, $S \in N$, $N \cap \Sigma = \varnothing$, $\Sigma \neq \varnothing$, $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ в нормальной форме Хомского. Требуется для заданного слова $w = w_0, \ldots, w_{n-1}$, $n \in \mathbb{N}$ определить, выводимо ли оно в этой грамматике.

Решение.

```
int n = len(w)
int m = size(N)
bool D[m][n+1][n+1] = false[m][n+1][n+1]
for int i = 0 .. n-1 do
    for (A -> a) in P do
        if w[i] = a then
            D[N.indexof(A)][i][i+1] = true
for int l=2 .. n do
    for int i=0 .. n-l do
        for int j=i+1 .. i+l-1 do
            for (A -> BC) in P do
                if D[N.indexof(B)][i][j] and D[N.indexof(C)][j][i+l] then
                    D[N.indexof(A)]][i][i+l] = true
return D[N.indexof(S)][0][n]
```

Тогда D[N.indexof(A)] $\iff A \vdash_G w_i \ldots w_{j-1}$.

6 Амортизационный анализ.

Амортизационный анализ — метод подсчёта времени, требуемого для выполнения последовательности операций над структурой данных. При этом время усредняется по всем выполняемым операциям, и анализируется средняя производительность операций в худшем случае.

Средняя амортизационная стоимость операций — величина a, находящаяся по формуле: $a = \dfrac{\sum_{i=1}^{n} t_n}{n}$, где $t_1, t_2, \ldots, t_n$ — время выполнения операций $1, 2, \ldots, n$, совершённых над структурой данных.

6.1 Метод усреднения.

В методе усреднения амортизационная стоимость операций определяется напрямую по формуле, указанной выше: суммарная стоимость всех операций алгоритма делится на их количество.

Пример. Рассмотрим стек с операцией multipop(a) – извлечение из стека a элементов. В худшем случае она работает за $O(n)$ времени, если удаляются все элементы массива.

Пусть n – количество операций, m – количество элементов, задействованных в этих операциях. Очевидно, $m \leq n$. Тогда:

$a = \frac{\sum\limits_{i=1}^{n} t_i}{n} = \frac{\sum\limits_{i=1}^{n}\sum\limits_{j=1}^{m} t_{ij}}{n} = \frac{\sum\limits_{j=1}^{m}\sum\limits_{i=1}^{n} t_{ij}}{n}$, где t_{ij} – стоимость i-ой операции над j-ым элементом. Величина $\sum\limits_{i=1}^{n} t_{ij}$ не превосходит 2, так как над элементом можно совершить только две операции, стоимость которых равна 1 – добавление и удаление. Тогда: $a \leq \frac{2m}{n} \leq 2$, так как $m \leq n$.

Таким образом, средняя амортизационная стоимость операций $a = O(1)$.

6.2 Метод потенциалов.

Теорема. Введём для каждого состояния структуры данных величину Φ – потенциал. Изначально потенциал равен Φ_0, а после выполнения i-й операции – Φ_i. Стоимость i-й операции обозначим $a_i = t_i + \Phi_i - \Phi_{i-1}$. Пусть n – количество операций, m – размер структуры данных. Тогда средняя амортизационная стоимость операций $a = O(f(n,m))$, если выполнены два условия:

1. Для любого $i : a_i = O(f(n,m))$
2. Для любого $i : \Phi_i = O(n \cdot f(n,m))$

$a = \frac{\sum\limits_{i=1}^{n} t_i}{n} = \frac{\sum\limits_{i=1}^{n} a_i + \sum\limits_{i=0}^{n-1} \Phi_i - \sum\limits_{i=1}^{n} \Phi_i}{n} = \frac{n \cdot O(f(n,m)) + \Phi_0 - \Phi_n}{n} = O(f(n,m))$

Пример. В качестве примера вновь рассмотрим стек с операцией multipop(a). Пусть потенциал – это количество элементов в стеке. Тогда:

1. Амортизационная стоимость операций
   (a) $a_{push} = 1 + 1 = 2$, так как время выполнения операции push – 1, и изменение потенциала – тоже 1.
   (b) $a_{pop} = 1 - 1 = 0$, так как время выполнения операции pop – 1, а изменение потенциала – -1.
   (c) $a_{multipop} = k - k = 0$, так как время выполнения операции multipop(k) – k, а изменение потенциала – $-k$.
2. Для любого $i : \Phi_i = O(n)$, так как элементов в стеке не может быть больше n.

Таким образом, $f(n,m) = 1$, а значит, средняя амортизационная стоимость операций $a = O(1)$.

6.3 Метод предоплаты.

Представим, что использование определённого количества времени равносильно использованию определённого количества монет (плата за выполнение каждой операции). В методе предоплаты каждому типу операций присваивается своя учётная стоимость. Эта стоимость может быть больше фактической, в таком случае лишние монеты используются как резерв для выполнения других операций в будущем, а может быть меньше, тогда гарантируется, что текущего накопленного резерва достаточно для выполнения операции. Для доказательства оценки средней амортизационной стоимости $O(f(n,m))$ нужно построить учётные стоимости так, что для каждой операции она будет составлять $O(f(n,m))$. Тогда для последовательности из n операций суммарно будет затрачено $n \cdot O(f(n,m))$ монет, следовательно, средняя амортизационная стоимость операций будет $a = \frac{\sum\limits_{i=1}^{n} t_i}{n} = \frac{n \cdot O(f(n,m))}{n} = O(f(n,m))$.

Пример. В качестве примера вновь рассмотрим стек с операцией multipop(a). При выполнении операции push будем использовать две монеты – одну для самой операции, а вторую – в качестве резерва. Тогда для операций pop и multipop учётную стоимость можно принять равной нулю и использовать для удаления элемента монету, оставшуюся после операции push.

Таким образом, для каждой операции требуется $O(1)$ монет, значит, средняя амортизационная стоимость операций $a = O(1)$.

7 RMQ.

Дан массив $A[1 \dots N]$ целых чисел. Поступают запросы вида (l, r), для каждого из которых требуется найти минимум среди элементов $A[l], A[l+1], \dots, A[r]$.

7.1 Naive solution.

Сделаем предподсчет минимума на всех отрезках за $O(n^2)$ и будем отвечать на запросы за $O(1)$.

7.2 Sparse Table.

Разреженная таблица — двумерная структура данных $ST[i][j]$, для которой выполнено следующее:

$$ST[i][j] = \min\left(A[i], A[i+1], \dots, A[i+2^j-1]\right), \quad j \in [0 \dots \log N].$$

Иначе говоря, в этой таблице хранятся минимумы на всех отрезках, длины которых равны степеням двойки. Объём памяти, занимаемый таблицей, равен $O(N \log N)$, и заполненными являются только те элементы, для которых $i + 2^j \leq N$.

Простой метод построения таблицы заключён в следующем реккурентном соотношении:

$$ST[i][j] = \begin{cases} \min\left(ST[i][j-1], ST[i+2^{j-1}][j-1]\right), & \text{если } j > 0; \\ A[i], & \text{если } j = 0; \end{cases}$$

Применение к RMQ. Предпосчитаем для длины отрезка l величину $\lfloor \log_2 l \rfloor$. Для этого введем функцию fl:

```
int fl(int len):
    if len = 1
        return 0
    else
        return fl(⌊len/2⌋) + 1
```

Вычисление $fl[l]$ происходит за $O(\log(l))$. А так как длина может принимать N различных значений, то суммарное время предпосчета составляет $O(N \log N)$.

Пусть теперь дан запрос (l, r). Заметим, что $\min(A[l], A[l+1], \ldots, A[r]) = \min\left(ST[l][j], ST[r-2^j+1][j]\right)$, где $j = \max\{k \mid 2^k \leq r-l+1\}$, то есть логарифм длины запрашиваемого отрезка, округленный вниз. Но эту величину мы уже предпосчитали, поэтому запрос выполняется за $O(1)$.

7.3 Дерево отрезков.

Для удобства дополним длину массива до степени двойки. В добавленные элементы массива допишем бесконечности. Дерево отрезков – это двоичное дерево, в каждой вершине которого написано значение заданной функции на некотором отрезке. Функция в нашем случае – это минимум.

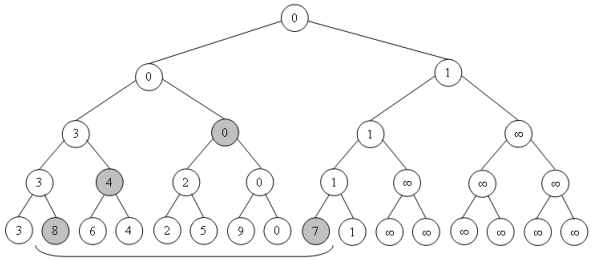
Каждому листу будет соответствовать элемент массива с номером, равным порядковому номеру листа в дереве. А каждой вершине, не являющейся листом, будет соответствовать отрезок из элементов массива соответствующих листам-потомкам этой вершины.

Построение дерева происходит за $O(n)$.

```
const int INF = INT_MAX;

void build_tree(const vector<int>& V) {
        int n = (1 << (log(n - 1) + 1));
        V.resize(2 * n, INF);

        for (int i = n; i < 2 * n; i++)
                V[i] = V[i - n];

        for (int i = n - 1; i > 0; i--)
                V[i] = min(V[2 * i], V[2 * i + 1]);
}
```

Запрос минимума.

Назовём фундаментальным отрезком в массиве такой отрезок, что существует вершина в дереве, которой он соответствует. Разобьём наш отрезк на минимальное количество непересекающихся фундаментальных. Покажем, что на каждом уровне их количество не превосходит 2.



Возьмём самый большой фундаментальный отрезок в разбиении. Пусть его длина – $2t$. Заметим, что фундаментальных отрезков длиной $2t$ – не более двух. Возьмём самый левый из имеющихся максимальных фундаментальных. Будем двигаться от него налево. Заметим, опять же, что длины отрезков будут убывать. Так же и с правым из максимальных. Тем самым получим, что фундаментальных отрезков – не более $2t$, что не превосходит $2\log(n)$.

Заведём два указателя – l и r, с помощью которых будем находить очередные фундаментальные отрезки разбиения. Изначально установим l и r указывающими на листы, соответствующие концам отрезка запроса. Заметим, что если l указывает на вершину, являющуюся правым сыном своего родителя, то эта вершина принадлежит разбиению на фундаментальные отрезки, в противном случае не принадлежит. Аналогично с указателем r – если он указывает на вершину, являющуюся левым сыном своего родителя, то добавляем её в разбиение. После этого сдвигаем оба указателя на уровень выше и повторяем операцию. Продолжаем операции пока указатели не зайдут один за другой.

Находя очередной фундаментальный отрезок, мы сравниваем минимум на нём с текущим найденным минимумом и уменьшаем его в случае необходимости. Асимптотика работы алгоритма – $O(\log(n))$, т.к. на каждом уровне мы выполняем константное число операций, а всего уровней – $\log(n)$.

```
int rmq_up(vector<int>& T, int l, int r) {
        int ans = INF;
        int n = T.size() / 2;
        l += n - 1, r += n - 1;
        while (l <= r) {
                if (l & 1)
                        ans = min(ans, T[l]);
                if (!(r & 1))
                        ans = min(ans, T[r]);
                l = (l + 1) / 2, r = (r - 1) / 2;
        }
        return ans;
}
```

Модификация. Для каждого листа есть ровно $\log(n)$ фундаментальных отрезков, которые его содержат – все они соответствуют вершинам, лежащим на пути от нашего листа до корня. Значит, при изменении элемента достаточно просто пробежаться от его листа до корня и обновить значение во всех вершинах на пути по формуле $T[i] = \min(T[2i], T[2i+1])$.

```
void update(vector<int>& T, int i, int x) {
        int n = T.size() / 2;
        i += n - 1;
        T[i] = x;
        while (i /= 2)
                T[i] = min(T[2 * i], T[2 * i + 1]);
}
```

8 LCA: сведение к RMQ и метод двоичного подъема.

Пусть дано корневое дерево T. На вход подаются запросы вида (u, v), для каждого запроса требуется найти их наименьшего общего предка.

Наименьшим общим предком двух узлов u и v в корневом дереве T называется узел w, который среди всех узлов, являющихся предками как узла u, так и v, имеет наибольшую глубину.

8.1 RMQ → LCA.

Будем решать задачу LCA, уже умея решать задачу RMQ. Тогда поиск наименьшего общего предка i-того и j-того элементов сводится к запросу минимума на отрезке массива, который будет введен позднее.

Препроцессинг. Для каждой вершины T определим глубину с помощью следующей рекурсивной формулы:

$$\text{depth}(u) = \begin{cases} 0 & u = \text{root}(T), \\ \text{depth}(v) + 1 & u = \text{son}(v). \end{cases}$$

Ясно, что глубина вершины элементарным образом поддерживается во время обхода в глубину.

Запустим обход в глубину из корня, который будет вычислять значения следующих величин:

1. Список глубин посещенных вершин d. Глубина текущей вершины добавляется в конец списка при входе в данную вершину, а также после каждого возвращения из её сына.

2. Список посещений узлов `vtx`, строящийся аналогично предыдущему, только добавляется не глубина а сама вершина.

3. Значение функции $\texttt{I}[u]$, возвращающей индекс в списке глубин d, по которому была записана глубина вершины u (например на момент входа в вершину).

Запрос. Будем считать, что $\text{rmq}(d, l, r)$ возвращает индекс минимального элемента в d на отрезке $[l..r]$. Тогда ответом на запрос $\text{lca}(u, v)$, где $\texttt{I}[u] \le \texttt{I}[v]$, будет $\texttt{vtx}[\text{rmq}(d, \texttt{I}[u], \texttt{I}[v])]$.

Теорема. Наименьшему общему предку вершин u, v соответствует минимальная глубина на отрезке $d[\texttt{I}[u], \texttt{I}[v]]$.

Рассмотрим два узла u, v корневого дерева T. Рассмотрим отрезок $d[\texttt{I}[u]..\texttt{I}[v]]$. Поскольку этот отрезок – путь из u в v, он проходит через их наименьшего общего предка w (в дереве есть только один простой путь между вершинами), а следовательно минимум на отрезке никак не больше глубины w. Заметим, что в момент добавления $\texttt{I}[u]$ обход посещал поддерево с корнем w. В момент добавления $\texttt{I}[v]$ мы все еще в поддереве с корнем w. Значит, и на отрезке между $\texttt{I}[u]$ и $\texttt{I}[v]$ мы находились внутри поддерева с корнем w. Отсюда сделаем заключение, что на рассматриваемом отрезке не посещалась вершина, отличная от w, с глубиной меньшей либо равной глубины w, т. к. подобной вершины нет в поддереве с корнем w.

Сложность. Прецпроцессинг – $O(n)$, запрос – зависит от реализации RMQ.

8.2 Метод двоичного подъема.

Препроцессинг. Заключается в том, чтобы посчитать функцию: $dp[v][i]$ – номер вершины, в которую мы придём если пройдём из вершины v вверх по подвешенному дереву 2^i шагов, причём если мы пришли в корень, то мы там и останемся. Для этого сначала обойдем дерево в глубину и для каждой вершины запишем номер её родителя $p[v]$ и глубину вершины в подвешенном дереве $d[v]$. Если v – корень, то $p[v] = v$. Тогда для функции dp есть рекуррентная формула: $dp[v][i] = \begin{cases} p[v] & i = 0, \\ dp[dp[v][i-1]][i-1] & i > 0. \end{cases}$

Для того чтобы отвечать на запросы нам нужны будут только те значения $dp[v][i]$, где $i \le \log_2 n$, ведь при больших i значение $dp[v][i]$ будет номером корня.

Всего состояний динамики $O(n \log n)$, где n – это количество вершин в дереве. Каждое состояние считается за $O(1)$. Поэтому суммарная сложность времени и памяти препроцессинга – $O(n \log n)$.

Запрос. Ответы на запросы будут происходить за время $O(\log n)$. Для ответа на запрос заметим сначала, что если $c = LCA(v, u)$, для некоторых v и u, то $d[c] \le \min(d[v], d[u])$. Поэтому если $d[v] < d[u]$, то пройдём от вершины u на $(d[u] - d[v])$ шагов вверх, это и будет новое значение u и это можно сделать за $O(\log n)$. Можно записать число $(d[u] - d[v])$ в двоичной системе, это представление этого число в виде суммы степеней двоек, $2^{i_1} + 2^{i_2} + \ldots + 2^{i_l}$ и для всех i_j пройти вверх последовательно из вершины u в $dp[u][i_j]$.

Дальше считаем, что $d[v] = d[u]$.

Если $v = u$, то ответ на запрос v.

А если $v \ne u$, то найдём такие вершины x и y, такие что $x \ne y$, x – предок v, y – предок u и $p[x] = p[y]$. Тогда ответом на запрос будет $p[x]$.

Научимся находить эти вершины x и y. Для этого сначала инициализируем $x = v$ и $y = u$. Дальше на каждом шаге находим такое максимальное k, что $dp[x][k] \ne dp[y][k]$. И проходим из вершин x и y на 2^k шагов вверх. Если такого k найти нельзя, то значения x и y, это те самые вершины, которые нам требуется найти, ведь $p[x] = dp[x][0] = dp[y][0] = p[y]$.

Оценим время работы. Заметим, что найденные k строго убывают. Во-первых, потому что мы находим на каждом шаге максимальное значение k, а во-вторых, два раза подряд мы одно и то же k получить не можем, так как тогда получилось бы, что можно пройти $2^k + 2^k = 2^{k+1}$ шагов, а значит вместо первого k, мы бы нашли $k+1$. А, значит, всего $O(\log n)$ значений k, их можно перебирать в порядке убывания. Сложность ответа на запрос $O(\log n)$.

9 Декартово дерево. Декартово дерево по неявному ключу.

9.1 Декартово дерево.

Декартово дерево – это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу.

Более строго, это бинарное дерево, в узлах которого хранятся пары (x, y), где x – это ключ, а y – это приоритет. Также оно является двоичным деревом поиска по x и пирамидой по y. Предполагая, что все x и все y являются различными, получаем, что если некоторый элемент дерева содержит (x_0, y_0), то у всех элементов в левом поддереве $x < x_0$, у всех элементов в правом поддереве $x > x_0$, а также и в левом, и в правом поддереве имеем: $y < y_0$.

9.2 Операции в декартовом дереве.

9.2.1 Spilt.

Операция split позволяет сделать следующее: разрезать исходное дерево T по ключу k. Возвращать она будет такую пару деревьев $\langle T_1, T_2\rangle$, что в дереве T_1 ключи меньше k, а в дереве T_2 все остальные: $\text{split}(T, k) \to \langle T_1, T_2\rangle$.

Эта операция устроена следующим образом.

Рассмотрим случай, в котором требуется разрезать дерево по ключу, большему ключа корня. Посмотрим, как будут устроены результирующие деревья T_1 и T_2:

1. T_1: левое поддерево T_1 совпадёт с левым поддеревом T. Для нахождения правого поддерева T_1, нужно разрезать правое поддерево T на T_1^R и T_2^R по ключу k и взять T_1^R.
2. T_2 совпадёт с T_2^R.

Случай, в котором требуется разрезать дерево по ключу, меньше либо равному ключа в корне, рассматривается симметрично.

Время работы. Оценим время работы операции split. Во время выполнения вызывается одна операция split для дерева хотя бы на один меньшей высоты и делается ещё $O(1)$ операций. Тогда итоговая трудоёмкость этой операции равна $O(h)$, где h – высота дерева.

9.2.2 Merge.

С помощью этой операции можно слить два декартовых дерева в одно. Причём, все ключи в первом дереве должны быть меньше, чем ключи во втором. В результате получается дерево, в котором есть все ключи из первого и второго деревьев: $\text{merge}(T_1, T_2) \to \{T\}$

Рассмотрим принцип работы этой операции. Пусть нужно слить деревья T_1 и T_2. Тогда, очевидно, у результирующего дерева T есть корень. Корнем станет вершина из T_1 или T_2 с наибольшим приоритетом y. Но вершина с самым большим y из всех вершин деревьев T_1 и T_2 может быть только либо корнем T_1, либо корнем T_2. Рассмотрим случай, в котором корень T_1 имеет больший y, чем корень T_2. Случай, в котором корень T_2 имеет больший y, чем корень T_1, симметричен этому.

Если y корня T_1 больше y корня T_2, то он и будет являться корнем. Тогда левое поддерево T совпадёт с левым поддеревом T_1. Справа же нужно подвесить объединение правого поддерева T_1 и дерева T_2.

Время работы. Рассуждая аналогично операции split, приходим к выводу, что трудоёмкость операции merge равна $O(h)$, где h – высота дерева.

9.2.3 Insert.

Операция $\text{insert}(T, k)$ добавляет в дерево T элемент k, где $k.x$ – ключ, а $k.y$ – приоритет.

Представим что элемент k, это декартово дерево из одного элемента, и для того чтобы его добавить в наше декартово дерево T, очевидно, нам нужно их слить. Но T может содержать ключи как меньше, так и больше ключа $k.x$, поэтому сначала нужно разрезать T по ключу $k.x$.

1. Реализация №1
   (a) Разобьём наше дерево по ключу, который мы хотим добавить, то есть $\text{split}(T, k.x) \to \langle T_1, T_2\rangle$.
   (b) Сливаем первое дерево с новым элементом, то есть $\text{merge}(T_1, k) \to T_1$.
   (c) Сливаем получившиеся дерево со вторым, то есть $\text{merge}(T_1, T_2) \to T$.
2. Реализация №2
   (a) Сначала спускаемся по дереву (как в обычном бинарном дереве поиска по $k.x$), но останавливаемся на первом элементе, в котором значение приоритета оказалось меньше $k.y$.
   (b) Теперь вызываем $\text{split}(T, k.x) \to \langle T_1, T_2\rangle$ от найденного элемента (от элемента вместе со всем его поддеревом)
   (c) Полученные T_1 и T_2 записываем в качестве левого и правого сына добавляемого элемента.
   (d) Полученное дерево ставим на место элемента, найденного в первом пункте.

9.2.4 Remove.

Операция $\text{remove}(T, x)$ удаляет из дерева T элемент с ключом x.

1. Реализация №1
   (a) Разобьём наше дерево по ключу, который мы хотим удалить, то есть $\text{split}(T, k.x) \to \langle T_1, T_2\rangle$.
   (b) Теперь отделяем от второго дерева элемент x, то есть самого левого ребёнка дерева T_2.
   (c) Сливаем первое дерево со вторым, то есть $\text{merge}(T_1, T_2) \to T$.
2. Реализация №2
   (a) Спускаемся по дереву (как в обычном бинарном дереве поиска по x), и ищем удаляемый элемент.
   (b) Найдя элемент, вызываем merge его левого и правого сыновей.

(c) Результат процедуры merge ставим на место удаляемого элемента.

Построение декартова дерева. Пусть нам известно из каких пар (x_i, y_i) требуется построить декартово дерево, причём также известно, что $x_1 < x_2 < \ldots < x_n$.

1. **Алгоритм за** $O(n \log n)$ Отсортируем все приоритеты по убыванию за $O(n \log n)$ и выберем первый из них, пусть это будет y_k. Сделаем (x_k, y_k) корнем дерева. Проделав то же самое с остальными вершинами получим левого и правого сына (x_k, y_k). В среднем высота Декартова дерева $\log n$ (см. далее) и на каждом уровне мы сделали $O(n)$ операций. Значит такой алгоритм работает за $O(n \log n)$.

2. **Алгоритм за** $O(n)$ Будем строить дерево слева направо, то есть начиная с (x_1, y_1) по (x_n, y_n), при этом помнить последний добавленный элемент (x_k, y_k). Он будет самым правым, так как у него будет максимальный ключ, а по ключам декартово дерево представляет собой [[Дерево поиска, наивная реализация|двоичное дерево поиска]]. При добавлении (x_{k+1}, y_{k+1}), пытаемся сделать его правым сыном (x_k, y_k), это следует сделать если $y_k > y_{k+1}$, иначе делаем шаг к предку последнего элемента и смотрим его значение y. Поднимаемся до тех пор, пока приоритет в рассматриваемом элементе меньше приоритета в добавляемом, после чего делаем (x_{k+1}, y_{k+1}) его правым сыном, а предыдущего правого сына делаем левым сыном (x_{k+1}, y_{k+1}). Заметим, что каждую вершину мы посетим максимум дважды: при непосредственном добавлении и, поднимаясь вверх (ведь после этого вершина будет лежать в чьём-то левом поддереве, а мы поднимаемся только по правому). Из этого следует, что построение происходит за $O(n)$.

9.3 Декартово дерево по неявному ключу.

Основная идея. Возьмем структуру данных динамический массив. В её стандартной реализации мы умеем добавлять элемент в конец вектора, узнавать значение элемента, стоящего на определенной позиции, изменять элемент по номеру и удалять последний элемент. Предположим, что нам необходима структура данных с вышеуказанными свойствами, а также с операциями: добавить элемент в любое место (с соответствующим изменением нумерации элементов) и удалить любой элемент (также с соответствующим изменением нумерации). Такую структуру можно реализовать на базе декартового дерева, результат часто называют декартово дерево по неявному ключу.

Ключ X. Как известно, декартово дерево – это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу. При реализации же декартова дерева по неявному ключу модифицируем эту структуру. А именно, оставим в нем только приоритет Y, а вместо ключа X будем использовать следующую величину: количество элементов в нашей структуре, находящихся левее нашего элемента. Иначе говоря, будем считать ключом порядковый номер нашего элемента в дереве, уменьшенный на единицу.

Заметим, что при этом сохранится структура двоичного дерева поиска по этому ключу (то есть модифицированное декартово дерево так и останется декартовым деревом). Однако, с этим подходом появляется проблема: операции добавления и удаления элемента могут поменять нумерацию, и при наивной реализации на изменение всех ключей потребуется $O(n)$ времени, где n – количество элементов в дереве.

Вспомогательная величина C. Решается эта проблема довольно просто. Основная идея заключается в том, что такой ключ X сам по себе нигде не хранится. Вместо него будем хранить вспомогательную величину C: количество вершин в поддереве нашей вершины (в поддерево включается и сама вершина). Обратим внимание, что все операции с обычным декартовым деревом делались сверху. Также заметим, что если по пути от корня до некой вершины просуммировать все такие величины в левых поддеревьях, в которые мы не пошли, увеличенные на единицу, то придя в саму вершину и добавив к этой величине количество элементов в её левом поддереве, мы получим как раз ее ключ X.

9.4 Операции, поддерживающие структуру декартова дерева.

Структура обычного декартова дерева поддерживается с помощью двух операций: split – разбиение одного декартова дерева на два таких, что в одном ключ X меньше, чем заданное значение, а в другом – больше, и merge – слияние двух деревьев, в одном из которых все ключи X меньше, чем во втором. С учетом отличий декартова дерева по неявному ключу от обычного, операции теперь будут описываться так: split(root, t) – разбиение дерева на два так, что в левом окажется ровно t вершин, и merge(root1, root) – слияние двух любых деревьев, соответственно.

9.4.1 Split.

Пусть процедура split запущена в корне дерева с требованием отрезать от дерева k вершин. Также известно, что в левом поддереве вершины находится l вершин, а в правом r. Рассмотрим все возможные случаи:

1. $l \geq k$. В этом случае нужно рекурсивно запустить процедуру split от левого сына с тем же параметром k. При этом новым левым сыном корня станет правая часть ответа рекурсивной процедуры, а правой частью ответа станет корень.

2. $l < k$ Случай симметричен предыдущему. Рекурсивно запустим процедуру split от правого сына с параметром $k - l - 1$. При этом новым правым сыном корня станет левая часть ответа рекурсивной процедуры, а левой частью ответа станет корень.

9.4.2 Merge.

Посмотрим на реализацию процедуры merge в обычном декартовом дереве. Заметим, что в ней программа ни разу не обращается к ключу X. Поэтому реализация процедуры merge для декартова дерева по неявному ключу вообще не будет отличаться от реализации той же процедуры в обычном декартовом дереве.

9.4.3 Поддержание корректности значений C.

Единственное действие, обеспечивающее корректность этих значений заключается в том, что после любого действия с детьми вершины нужно записать в ее поле C сумму этих значений в ее новых детях, увеличенную на единицу.

10 Минимальное остовное дерево: алгоритмы Прима и Крускала.

10.1 Остовные деревья.

Рассмотрим связный неориентированный взвешенный граф $G = (V, E)$, где V – множество вершин, E – множество ребер. Вес ребра определяется, как функция $w : E \to \mathbb{R}$.

Остовное дерево графа $G = (V, E)$ – ациклический связный подграф данного связного неориентированного графа.

Минимальное остовное дерево графа $G = (V, E)$ – это его ациклический связный подграф, в который входят все его вершины, обладающий минимальным суммарным весом ребер.

Пусть G' – подграф некоторого минимального остовного дерева графа $G = (V, E)$.

Ребро $(u, v) \notin G'$ называется безопасным, если при добавлении его в G', $G' \cup \{(u, v)\}$ также является подграфом некоторого минимального остовного дерева графа G.

Разрезом неориентированного графа $G = (V, E)$ называется разбиение V на два непересекающихся подмножества: S и $T = V \setminus S$. Обозначается как $\langle S, T\rangle$.

Ребро $(u, v) \in E$ пересекает разрез $\langle S, T\rangle$, если один из его концов принадлежит множеству S, а другой – множеству T.

Лемма о безопасном ребре. Рассмотрим связный неориентированный взвешенный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$. Пусть $G' = (V, E')$ – подграф некоторого минимального остовного дерева G, $\langle S, T\rangle$ – разрез G, такой, что ни одно ребро из E' не пересекает разрез, а (u, v) – ребро минимального веса среди всех ребер, пересекающих разрез $\langle S, T\rangle$. Тогда ребро $e = (u, v)$ является безопасным для G'.

Достроим E' до некоторого минимального остовного дерева, обозначим его T_{min}. Если ребро $e \in T_{min}$, то лемма доказана, поэтому рассмотрим случай, когда ребро $e \notin T_{min}$. Рассмотрим путь в T_{min} от вершины u до вершины v. Так как эти вершины принадлежат разным долям разреза, то хотя бы одно ребро пути пересекает разрез, назовем его e'. По условию леммы $w(e) \leq w(e')$. Заменим ребро e' в T_{min} на ребро e. Полученное дерево также является минимальным остовным деревом графа G, поскольку все вершины G по-прежнему связаны и вес дерева не увеличился. Следовательно $E' \cup \{e\}$ можно дополнить до минимального остовного дерева в графе G, то есть ребро e – безопасное.

10.2 Алгоритм Прима.

Будем последовательно строить поддерево F ответа в графе G, поддерживая приоритетную очередь Q из вершин $G \setminus F$, в которой ключом для вершины v является $\min\limits_{u \in V(F), uv \in E(G)} w(uv)$ — вес минимального ребра из вершин F в вершины $G \setminus F$. Также для каждой вершины в очереди будем хранить $p(v)$ — вершину u, на которой достигается минимум в определении ключа. Дерево F поддерживается неявно, и его ребра — это пары $(v, p(v))$, где $v \in G \setminus \{r\} \setminus Q$, а r — корень F. Изначально F пусто и значения ключей у всех вершин равны $+\infty$. Выберём произвольную вершину r и присвоим её ключу значение 0. На каждом шаге будем извлекать минимальную вершину v из приоритетной очереди и релаксировать все ребра vu, такие что $u \in Q$, выполняя при этом операцию decreaseKey над очередью и обновление $p(v)$. Ребро $(v, p(v))$ при этом добавляется к ответу.

```
function primFindMST():
    for v ∈ V(G)
        key[v] = ∞
        p[v] = null
    r = random vertex of G
    key[r] = 0
    Q.push(V(G))
    while not Q.isEmpty()
        v = Q.extractMin()
        for vu ∈ E(G)
            if u ∈ Q and key[u] > w(v, u)
                p[u] = v
                key[u] = w(v, u)
                Q.decreaseKey(u, key[u])
```

Ребра дерева восстанавливаются из его неявного вида после выполнения алгоритма. Если делать с бинарной кучей, то вместо операции decreaseKey, будем всегда просто добавлять вершину с новым ключом. Если из кучи достали вершину с ключом, значение которого больше чем у нее уже стоит – просто игнорировать. Вершин в куче будет не больше n^2, поэтому операция extractMin будет выполняться за $O(\log n)$. Максимальное количество пересчетов, которое мы проделаем, равняется количеству ребер, то есть m, поэтому общая асимптотика составит $O(m \log n)$, что хорошо только на разреженных графах.

Корректность. По поддерживаемым инвариантам после извлечения вершины v ($v \neq r$) из Q ребро $(v, p(v))$ является ребром минимального веса, пересекающим разрез (F, Q). Значит, по лемме о безопасном ребре, оно безопасно. Алгоритм построения MST, добавляющий безопасные ребра, причём делающий это ровно $|V| - 1$ раз, корректен.

10.3 Алгоритм Крускала.

Будем последовательно строить подграф F графа G, пытаясь на каждом шаге достроить F до некоторого MST. Начнем с того, что включим в F все вершины графа G. Теперь будем обходить множество $E(G)$ в порядке неубывания весов ребер. Если очередное ребро e соединяет вершины одной компоненты связности F, то добавление его в остов приведет к возникновению цикла в этой компоненте связности. В таком случае, очевидно, e не может быть включено в F. Иначе e соединяет разные компоненты связности F, тогда существует $\langle S, T\rangle$ разрез такой, что одна из компонент связности составляет одну его часть, а оставшаяся часть графа — вторую. Тогда e — минимальное ребро, пересекающее этот разрез. Значит, из леммы о безопасном ребре следует, что e является безопасным, поэтому добавим это ребро в F. На последнем шаге ребро соединит две оставшиеся компоненты связности, полученный подграф будет минимальным остовным деревом графа G. Для проверки возможности добавления ребра используется система непересекающихся множеств.

```
function kruskalFindMST():
    F ← V(G)
    sort(E(G))
    for vu ∈ E(G)
        if v and u in different connected components
            F = F⋃vu
    return F
```

Ассимптотика. Сортировка E займет $O(E \log E)$. Работа с СНМ займет $O(E\alpha(V))$, где α — обратная функция Аккермана, которая не превосходит 4 во всех практических приложениях и которую можно принять за константу. Алгоритм работает за $O(E(\log E + \alpha(V))) = O(E \log E)$.

11 Максимальные потоки в сети. Методы: Форда-Фалкерсона; Эдмондса-Карпа (б/д).

11.1 Максимальные потоки в сети.

Сетью называется ориентированный граф $G = (V, E)$, для которого задана функция $c\colon V \times V \to [0; +\infty)$ — пропускная способность ребра, причем $(u, v) \in E \iff c(u, v) > 0$. Обычно для сети задаются также исток $s \in V$ и сток $t \in V \setminus \{s\}$.

Потоком f в сети $G = (V, E)$ называется функция $f\colon V \times V \to \mathbb{R}$, удовлетворяющая условиям:

1. $\forall u, v \in V(f(u, v) = -f(v, v))$ (антисимметричность);
2. $\forall u, v \in V(|f(u, v)| \leqslant c(u, v))$ (ограничение пропускной способности);
3. $\forall u \in V \setminus \{s, t\}(\sum_{v \in V} f(u, v) = 0)$ (закон сохранения потока).

Величиной потока f в сети $G = (V, E)$ называется число $|f| = \sum_{v \in V} f(s, v) = \sum_{u \in V} f(u, t)$ (корректность определения очевидна из закона сохранения потока).

Разрезом (S, T) в сети $G = (V, E)$ называется такое разбиение множества вершин на два непустых подмножества, что $s \in S$, $t \in T$. Пропускной способностью разреза называется $c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$. Минимальным разрезом называется разрез, пропускная способность которого минимальна среди всех разрезов сети.

Пусть f — поток в ести $G = (V, E)$. Тогда остаточной пропускной способностью называется функция $c_f = c - f$. Сеть $G_f = (V, E)$ с пропускной способностью c_f называется остаточной сетью.

Будем говорить что в сети $G = (V, E)$ есть ребро (u, v), если $c(u, v) > 0$, и есть путь из u_0 в u_n, если существует такая последовательность вершин $u_1, \ldots, u_{n-1}$, что для любого $k \in [n]$ есть ребро (u_{k-1}, u_k). Увеличивающими путями называются пути из s в t.

11.2 Метод Форда-Фалкерсона.

Теорема (Форда-Фалкерсона). Пусть $G = (V, E)$ — сеть, f — поток в ести G. Тогда следующие условия равносильны:

1. f — максимальный поток;
2. остаточная сеть G_f не содержит увеличивающих путей;
3. $|f| = c(S, T)$ для некоторого минимального разреза (S, T).

Доказательство.

1. 1. $\Rightarrow$ 2. Очевидно.
2. 2. $\Rightarrow$ 3. Положим $S = \{v \in V : s \rightsquigarrow v\}$ и $T = V \setminus T$. Тогда $s \in S$, $t \in T$, а для любых $u \in S$, $v \in T$ верно $f(u, v) = c(u, v)$ (иначе через это ребро можно дойти из u в v, а значит, v была бы достижима из s). Таким образом, (S, T) — разрез. Тогда
   $|f| =$
   $= \sum_{v \in V} f(s, v) =$
   $= \sum_{v \in V} f(s, v) + 0 =$
   $= \sum_{v \in V} f(s, v) + \sum_{u \in S - \{s\}} \sum_{v \in V} f(u, v) =$
   $= \sum_{u \in S} \sum_{v \in V} f(u, v) =$
   $= \sum_{u \in S} \sum_{v \in V} f(u, v) - 0 =$
   $= \sum_{u \in S} \sum_{v \in V} f(u, v) - \sum_{u \in S} \sum_{v \in S} f(u, v) =$
   $= \sum_{u \in S} \sum_{v \in T} f(u, v) =$
   $= \sum_{u \in S} \sum_{v \in T} c(u, v) =$
   $= c(S, T)$.
3. 3. $\Rightarrow$ 1. Достаточно доказать, что $|f| \leqslant c(S', T')$ для любого минимального разреза (S', T'). Действительно, как выяснено выше, $|f| = \sum_{v \in S'} \sum_{u \in T'} f(u, v) \leqslant \sum_{v \in S'} \sum_{u \in T'} c(u, v) = c(S', T')$. Значит, величина потока не превосходит пропускной способности всех минимальных разрезов. Но так как она равна одной из них, то значит, величина потока максимальная.

Псевдокод.

```
FordFalkersonBase(G):
    for (u, v) in E do
        (u, v).flow = 0
    while exists augmenting path p in G.f do
        c.f(p) = min{ c(u, v) | (u, v) in p }
        // remove this augmenting path
        for (u, v) in p do
            (u, v).flow += c.f(p)
            (v, u).flow -= c.f(p)
    // there is no augmenting paths, so theorem is applicable
    S = { u | s -> u }
    T = V - S
    return c(S, T)
```

Если указанный алгоритм завершает работу, то найденный поток максимален. Однако он может работать бесконечно долго, причем последовательность промежуточных потоков может не сходиться к величине максимального потока. Доказано, что метод завершается, если все пропускные способности целые, одгнако и он может работаь долго.

Время работы алгоритма для случая целочисленных пропускных способностей. Положим $C = \max\{c(e) : e \in E\}$, а через F обозначим величину максимального потока. Очевидно, $F \leqslant CE$ (количество ребер на максимальную пропускную способность) — это оценка на количество внешних циклов, а внутренний цикл работает за $O(E)$. Таким образом, суммарное время работы — $O(CE^2)$. Также можно показать, что количество увеличивающих путей не превосходит $E \log C$, что улучшает оценку до $O(E^2 \log C)$.

11.3 Метод Эдмондса-Карпа.

Алгоритм Эдмондса-Карпа заключается в том, что в FordFalkersonBase среди всех увеличивающих путей выбирается кратчайший по количетсву ребер (выбирается с помощью обхода в ширину, так что не влияет на асимптотику). Доказывается, что в таком случае количество ребер на каждой итерации уменьшается хотя бы на 2, таким образом, алгоритм совершает не более $\frac{1}{2}V + 1$ итерации, что дает ему время работы $O(VE^2)$.

12 Обход графа в глубину, ширину.

12.1 Обход в ширину.

Пусть задан невзвешенный ориентированный граф $G = (V, E)$, в котором выделена исходная вершина s. Требуется найти длину кратчайшего пути (если таковой имеется) от одной заданной вершины до другой.

Для алгоритма потребуются очередь и множество посещенных вершин was, которые изначально содержат одну вершину s. На каждом шагу алгоритм берет из начала очереди вершину v и добавляет все непосещенные смежные с v вершины в was и в конец очереди. Если очередь пуста, то алгоритм завершает работу.

```
int BFS(G: (V, E), source: int, destination: int):
    d = int[|V|]
    fill(d, ∞)
    d[source] = 0
    Q = ∅
    Q.push(source)
    while Q ≠ ∅
        u = Q.pop()
        for v: (u, v) in E
        if d[v] == ∞
            d[v] = d[u] + 1
            Q.push(v)
    return d[destination]
```

Утверждение. В очереди поиска в ширину расстояние вершин до s монотонно неубывает.

Докажем это утверждение индукцией по числу выполненных алгоритмом шагов.

Введем дополнительный инвариант: у любых двух вершин из очереди, расстояние до s отличается не более чем на 1.

База: изначально очередь содержит только одну вершину s.

Переход: пусть после i-й итерации в очереди $a + 1$ вершин с расстоянием x и b вершин с расстоянием $x + 1$.

Рассмотрим $i+1$-ю итерацию. Из очереди достаем вершину v, с расстоянием x. Пусть у v есть r непосещенных смежных вершин. Тогда, после их добавления, в очереди находится a вершин с расстоянием x и, после них, $b + r$ вершин с расстоянием $x + 1$.

Оба инварианта сохранились $\Rightarrow$ после любого шага алгоритма элементы в очереди неубывают.

Теорема. Алгоритм поиска в ширину в невзвешенном графе находит длины кратчайших путей до всех достижимых вершин.

Допустим, что это не так. Выберем из вершин, для которых кратчайшие пути от s найдены некорректно, ту, настоящее расстояние до которой минимально. Пусть это вершина u, и она имеет своим предком в дереве обхода в ширину v, а предок в кратчайшем пути до u — вершина w.

Так как w — предок u в кратчайшем пути, то $\rho(s, u) = \rho(s, w) + 1 > \rho(s, w)$, и расстояние до w найдено верно, $\rho(s, w) = d[w]$. Значит, $\rho(s, u) = d[w] + 1$.

Так как v — предок u в дереве обхода в ширину, то $d[u] = d[v] + 1$.

Расстояние до u найдено некорректно, поэтому $\rho(s, u) < d[u]$. Подставляя сюда два последних равенства, получаем $d[w] + 1 < d[v] + 1$, то есть, $d[w] < d[v]$. Из ранее доказанной леммы следует, что в этом случае вершина w попала в очередь и была обработана раньше, чем v. Но она соединена с u, значит, v не может быть предком u в дереве обхода в ширину, мы пришли к противоречию, следовательно, найденные расстояния до всех вершин являются кратчайшими.

Оценка сложности. Оценим время работы для входного графа $G = (V, E)$, где множество ребер E представлено списком смежности. В очередь добавляются только непосещенные вершины, поэтому каждая вершина посещается не более одного раза. Операции внесения в очередь и удаления из нее требуют $O(1)$ времени, так что общее время работы с очередью составляет $O(|V|)$ операций. Для каждой вершины v рассматривается не более $\deg(v)$ ребер, инцидентных ей. Так как $\sum\limits_{v \in V} \deg(v) = 2|E|$, то время, используемое на работу с ребрами, составляет $O(|E|)$. Поэтому общее время работы алгоритма поиска в ширину — $O(|V| + |E|)$.

12.2 Обход в глубину.

Общая идея алгоритма состоит в следующем: для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них.

Пошаговое представление.

1. Выбираем любую вершину из еще не пройденных, обозначим ее как u.
2. Запускаем процедуру dfs(u).
   (a) Помечаем вершину u как пройденную.
   (b) Для каждой не пройденной смежной с u вершиной (назовем ее v) запускаем $dfs(v)$.
3. Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

```
function doDfs(G[n]: Graph):
    visited = array[n, false]

    function dfs(u: int):
        visited[u] = true
        for v: (u, v) in G
            if not visited[v]
            dfs(v)

    for i = 1 to n
        if not visited[i]
            dfs(i)
```

Время работы. Оценим время работы обхода в глубину. Процедура dfs вызывается от каждой вершины не более одного раза, а внутри процедуры рассматриваются все такие ребра $\{e \mid \text{begin(e)} = u\}$. Всего таких ребер для всех вершин в графе $O(E)$, следовательно, время работы алгоритма оценивается как $O(V + E)$.

13 Поиск кратчайших путей в графе: алгоритмы Дейкстры, Форда-Беллмана, Флойда-Уоршелла.

13.1 Алгоритм Дейкстры.

В ориентированном взвешенном графе $G = (V, E)$, вес рёбер которого неотрицателен и определяется весовой функцией $w : E \to \mathbb{R}$, алгоритм Дейкстры находит длины кратчайших путей из заданной вершины s до всех остальных.

В алгоритме поддерживается множество вершин U, для которых уже вычислены длины кратчайших путей до них из s. На каждой итерации основного цикла выбирается вершина $u \notin U$, которой на текущий момент соответствует минимальная оценка кратчайшего пути. Вершина u добавляется в множество U и производится релаксация всех исходящих из неё рёбер.

```
func dijkstra(s):
    for v ∈ V
        d[v] = ∞
        used[v] = false
    d[s] = 0
    for i ∈ V
        v = null
        for j ∈ V
            if !used[j] and (v == null or d[j] < d[v])
                v = j
        if d[v] == ∞
            break
        used[v] = true
        for e : edges going from v
            if d[v] + e.len < d[e.to]
                d[e.to] = d[v] + e.len
```

Теорема. Пусть $G = (V, E)$ – ориентированный взвешенный граф, вес рёбер которого неотрицателен, s – стартовая вершина. Тогда после выполнения алгоритма Дейкстры $d(u) = \rho(s, u)$ для всех u, где $\rho(s, u)$ – длина кратчайшего пути из вершины s в вершину u.

Доказательство. Докажем по индукции, что в момент посещения любой вершины u, $d(u) = \rho(s, u)$.

1. На первом шаге выбирается s, для неё выполнено: $d(s) = \rho(s, s) = 0$.

2. Пусть для n первых шагов алгоритм сработал верно и на $n + 1$ шагу выбрана вершина u. Докажем, что в этот момент $d(u) = \rho(s, u)$. Для начала отметим, что для любой вершины v, всегда выполняется $d(v) \geqslant \rho(s, v)$ (алгоритм не может найти путь короче, чем кратчайший из всех существующих). Пусть P — кратчайший путь из s в u, v – первая непосещённая вершина на P, z – предшествующая ей (следовательно, посещённая). Поскольку путь P кратчайший, его часть, ведущая из s через z в v, тоже кратчайшая, следовательно $\rho(s, v) = \rho(s, z) + w(zv)$. По предположению индукции, в момент посещения вершины z выполнялось $d(z) = \rho(s, z)$, следовательно, вершина v тогда получила метку не больше чем $d(z) + w(zv) = \rho(s, z) + w(zv) = \rho(s, v)$, следовательно, $d(v) = \rho(s, v)$. С другой стороны, поскольку сейчас мы выбрали вершину u, её метка минимальна среди непосещённых, то есть $d(u) \leq d(v) = \rho(s, v) \leq \rho(s, u)$, где второе неравенсто верно из-за ранее упомянутого определения вершины v в качестве первой непосещённой вершины на P, то есть вес пути до промежуточной вершины не превосходит веса пути до конечной вершины вследствие неотрицательности весовой функции. Комбинируя это с $d(u) \geq \rho(s, u)$, имеем $d(u) = \rho(s, u)$, что и требовалось доказать.

Поскольку алгоритм заканчивает работу, когда все вершины посещены, в этот момент $d(u) = \rho(s, u)$ для всех u.

Оценка сложности

В реализации алгоритма присутствует функция выбора вершины с минимальным значением d и релаксация по всем рёбрам для данной вершины. Асимптотика работы зависит от реализации.

Пусть n – количество вершин в графе, m – количество рёбер в графе.

	Поиск минимума	Релаксация	Общее	Комментарий
Наивная реализация:	$O(n)$	$O(1)$	$O(n^2+m)$	n раз осуществляем поиск вершины с минимальной величиной d среди $O(n)$ непомеченных вершин и m раз проводим релаксацию за $O(1)$. Для плотных графов ($m \approx n^2$) данная асимптотика является оптимальной.
Двоичная куча:	$O(\log n)$	$O(\log n)$	$O(m \log n)$	Используя двоичную кучу можно выполнять операции извлечения минимума и обновления элемента за $O(\log n)$. Тогда время работы алгоритма Дейкстры составит $O(n \log n + m \log n) = O(m \log n)$.

Изначально поместим в контейнер стартовую вершину s. Основной цикл будет выполняться, пока в контейнере есть хотя бы одна вершина. На каждой итерации извлекается вершина с наименьшим расстоянием d и выполняются релаксации по рёбрам из неё. При выполнении успешной релаксации нужно удалить из контейнера вершину, до которой обновляем расстояние, а затем добавить её же, но с новым расстоянием.

В обычных кучах нет операции удаления произвольного элемента. При релаксации можно не удалять старые пары, в результате чего в куче может находиться одновременно несколько пар расстояние-вершина для одной вершины (с разными расстояниями). Для корректной работы при извлечении из кучи будем проверять расстояние: пары, в которых расстояние отлично от $d[v]$ будем игнорировать. При этом асимптотика будет $O(m \log m)$ вместо $O(m \log n)$.

13.2 Алгоритм Форда-Беллмана.

Для заданного взвешенного графа $G = (V, E)$ найти кратчайшие пути из заданной вершины s до всех остальных вершин. В случае, когда в графе G содержатся отрицательные циклы, достижимые из s, сообщить, что кратчайших путей не существует.

```
bool fordBellman(s):
    for v ∈ V
        d[v] = ∞
    d[s] = 0
    for i = 0 to |V| − 1
        for (u, v) ∈ E
            if d[v] > d[u] + ω(u, v)
                d[v] = d[u] + ω(u, v)
    for (u, v) ∈ E
        if d[v] > d[u] + ω(u, v)
            return false
    return true
```

В этом алгоритме используется релаксация, в результате которой $d[v]$ уменьшается до тех пор, пока не станет равным $\delta(s, v)$.
$d[v]$ – оценка веса кратчайшего пути из вершины s в каждую вершину $v \in V$.

$\delta(s, v)$ – фактический вес кратчайшего пути из s в вершину v.

Лемма. Пусть $G = (V, E)$ – взвешенный ориентированный граф, s – стартовая вершина. Тогда после завершения $|V| - 1$ итераций цикла для всех вершин, достижимых из s, выполняется равенство $d[v] = \delta(s, v)$.

Доказательство. Рассмотрим произвольную вершину v, достижимую из s. Пусть $p = \langle v_0, ..., v_k \rangle$, где $v_0 = s$, $v_k = v$ – кратчайший ациклический путь из s в v. Путь p содержит не более $|V| - 1$ ребер. Поэтому $k \leq |V| - 1$.

Докажем следующее утверждение:

После n $(n \leq k)$ итераций первого цикла алгоритма, $d[v_n] = \delta(s, v_n)$. Воспользуемся индукцией по n:

База индукции: Перед первой итерацией утверждение очевидно выполнено: $d[v_0] = d[s] = \delta(s, s) = 0$.

Индукционный переход: Пусть после n $(n < k)$ итераций, верно что $d[v_n] = \delta(s, v_n)$. Так как (v_n, v_{n+1}) принадлежит кратчайшему пути от s до v, то $\delta(s, v_{n+1}) = \delta(s, v_n) + \omega(v_n, v_{n+1})$. Во время $l + 1$ итерации релаксируется ребро (v_n, v_{n+1}), следовательно по завершению итерации будет выполнено: $d[v_{n+1}] \leq d[v_n] + \omega(v_n, v_{n+1}) = \delta(s, v_n) + \omega(v_n, v_{n+1}) = \delta(s, v_{n+1})$. Ясно, что $d[v_{n+1}] \geq \delta(s, v_{n+1})$, поэтому верно что после $l + 1$ итерации $d[v_{n+1}] = \delta(s, v_{n+1})$.

Итак, выполнены равенства $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$.

Теорема. Пусть $G = (V, E)$ – взвешенный ориентированный граф, s – стартовая вершина. Если граф G не содержит отрицательных циклов, достижимых из вершины s, то алгоритм возвращает $true$ и для всех $v \in V$ $d[v] = \delta(s, v)$. Если граф G содержит отрицательные циклы, достижимые из вершины s, то алгоритм возвращает $false$.

Доказательство. Пусть граф G не содержит отрицательных циклов, достижимых из вершины s.

Тогда если вершина v достижима из s, то по лемме $d[v] = \delta(s, v)$. Если вершина v не достижима из s, то $d[v] = \delta(s, v) = \infty$ из несуществования пути.

Теперь докажем, что алгоритм вернет значение $true$.

После выполнения алгоритма верно, что для всех $(u, v) \in E$, $d[v] = \delta(s, v) \leq \delta(s, u) + \omega(u, v) = d[u] + \omega(u, v)$, значит ни одна из проверок не вернет значения $false$.

Пусть граф G содержит отрицательный цикл $c = v_0, ..., v_k$, где $v_0 = v_k$, достижимый из вершины s. Тогда $\sum\limits_{i=1}^{k} \omega(v_{i-1}, v_i) < 0$.

Предположим, что алгоритм возвращает $true$, тогда для $i = 1, ..., k$ выполняется $d[v_i] \leq d[v_{i-1}] + \omega(v_{i-1}, v_i)$.

Просуммируем эти неравенства по всему циклу: $\sum\limits_{i=1}^{k} d[v_i] \leq \sum\limits_{i=1}^{k} d[v_{i-1}] + \sum\limits_{i=1}^{k} \omega(v_{i-1}, v_i)$.

Из того, что $v_0 = v_k$ следует, что $\sum\limits_{i=1}^{k} d[v_i] = \sum\limits_{i=1}^{k} d[v_{i-1}]$.

Получили, что $\sum\limits_{i=1}^{k} \omega(v_{i-1}, v_i) \geq 0$, что противоречит отрицательности цикла c.

Сложность. Инициализация занимает $\Theta(V)$ времени, каждый из $|V| - 1$ проходов требует $\Theta(E)$ времени, обход по всем ребрам для проверки наличия отрицательного цикла занимает $O(E)$ времени. Значит алгоритм Беллмана-Форда работает за $O(VE)$ времени.

13.3 Алгоритм Флойда-Уоршелла.

Дан взвешенный ориентированный граф $G(V, E)$, в котором вершины пронумерованы от 1 до n.

$$\omega_{uv} = \begin{cases} \text{weight of } uv, & \text{if } uv \in E \\ +\infty, & \text{if } uv \notin E \end{cases}$$

Требуется найти матрицу кратчайших расстояний d, в которой элемент d_{ij} либо равен длине кратчайшего пути из i в j, либо равен $+\infty$, если вершина j не достижима из i.

Первое приближение. Обозначим длину кратчайшего пути между вершинами u и v, содержащего, помимо u и v, только вершины из множества $\{1..i\}$ как $d^{(i)}_{uv}$, $d^{(0)}_{uv} = \omega_{uv}$.

На каждом шаге алгоритма, мы будем брать очередную вершину (пусть её номер – i) и для всех пар вершин u и v вычислять $d^{(i)}_{uv} = \min(d^{(i-1)}_{uv}, d^{(i-1)}_{ui} + d^{(i-1)}_{iv})$. То есть, если кратчайший путь из u в v, содержащий только вершины из множества $\{1..i\}$, проходит через вершину i, то кратчайшим путем из u в v является кратчайший путь из u в i, объединенный с кратчайшим путем из i в i. В противном случае, когда этот путь не содержит вершины i, кратчайший путь из u в v, содержащий только вершины из множества $\{1..i\}$ является кратчайшим путем из u в v, содержащим только вершины из множества $\{1..i-1\}$.

$d^{(0)}_{uv} = w$
for $i \in V$
  for $u \in V$
    for $v \in V$
      $d^{(i)}_{uv} = \min(d^{(i-1)}_{uv}, d^{(i-1)}_{ui} + d^{(i-1)}_{iv})$

В итоге получаем, что матрица $d^{(n)}$ и является искомой матрицей кратчайших путей, поскольку содержит в себе длины кратчайших путей между всеми парами вершин, имеющих в качестве промежуточных вершин вершины из множества $\{1..n\}$, что есть попросту все вершины графа. Такая реализация работает за $\Theta(n^3)$ времени и использует $\Theta(n^3)$ памяти.

Финальная версия. Утверждается, что можно избавиться от одной размерности в массиве d, т.е. использовать двумерный массив d_{uv}. В процессе работы алгоритма поддерживается инвариант $\rho(u, v) \le d_{uv} \le d^{(i)}_{uv}$, а, поскольку, после выполнения работы алгоритма $\rho(u, v) = d^{(n)}_{uv}$, то тогда будет выполняться и $\rho(u, v) = d_{uv}$.

Утверждение. В течение работы алгоритма Флойда выполняются неравенства: $\rho(u, v) \le d_{uv} \le d^{(i)}_{uv}$.

Доказательство. После инициализации все неравенства, очевидно, выполняются.

Докажем второе неравенство индукцией по итерациям алгоритма.

Пусть также d'_{uv} – значение d_{uv} сразу после $i-1$ итерации.

Покажем, что $d_{uv} \le d^{(i)}_{uv}$, зная, что $d'_{uv} \le d^{(i-1)}_{uv}$.

Рассмотрим два случая:

1. Значение $d^{(i)}_{uv}$ стало меньше, чем $d^{(i-1)}_{uv}$. Тогда $d^{(i)}_{uv} = d^{(i-1)}_{ui} + d^{(i-1)}_{iv} \ge$ (выполняется на шаге $i-1$, по индукционному предположению) $\ge d'_{ui} + d'_{iv} \ge d_{uv}$.

2. В ином случае всё очевидно: $d^{(i)}_{uv} = d^{(i-1)}_{uv} \ge d'_{uv} \geqslant d_{uv}$, и неравенство тривиально.

Докажем первое неравенство от противного.

Пусть неравенство было нарушено, рассмотрим момент, когда оно было нарушено впервые. Пусть это была i-ая итерация и в этот момент изменилось значение d_{uv} и выполнилось $\rho(u, v) > d_{uv}$. Так как d_{uv} изменилось, то $d_{uv} = d_{ui} + d_{iv} \ge$ (так как ранее $\forall u, v \in V : \rho(u, v) \le d_{uv}$) $\ge \rho(u, i) + \rho(i, v) \ge$ (по неравенству треугольника) $\ge \rho(u, v)$. Итак $d_{uv} \ge \rho(u, v)$ – противоречие.

```
func floyd(w):
    d = ω
    for i ∈ V
        for u ∈ V
            for v ∈ V
                d[u][v] = min(d[u][v], d[u][i] + d[i][v])
```

Данная реализация работает за время $\Theta(n^3)$, но требует уже $\Theta(n^2)$ памяти.

14 Поиск сильносвязных компонент в графе.

Пусть дан ориентированный граф $G = (V, E)$, $E \subseteq V \times V$. Тогда отношение $R \subseteq V \times V$ называется сильной связностью, если $(u, v) \in R \iff (u \rightsquigarrow v) \wedge (v \rightsquigarrow u)$.

Теорема. Сильная связность является отношением эквивалентности.

Доказательство. Очевидно.

Классы эквивалентности вершин графа по сильной связности называются сильносвязными компонентами. Неформально, это такие наибольшие подмножества подмножества вершин графа, что из каждой вершины есть путь в любую другую.

Алгоритм нахождения компонент сильной связности. На входе ориентированный граф $G = (V, E)$, $E \subseteq V \times V$.

1. Построить граф $H = (V, E')$, где $E' = \{(u, v) : (v, u) \in E\}$, — граф с инвертированными ребрами.

2. Выполнить обход в глубину графа H, запомнив для каждой вершины f_u — время выхода из вершины.

3. Выполнить обход в глубину графа G, перебирая вершины в порядке убывания f_u. На этом шаге будут построены деревья обхода в глубину, вершины из этих деревьев и будут образовывать сильносвязные компоненты.

Теорема (корректность алгоритма). Пусть $G = (V, E)$, $E \subseteq V \times V$ — орграф. Тогда вершины $s, t \in V$ принадлежат одной сильносвязной компоненте $\iff$ они окажутся в одном дереве обхода в глубину по алгоритму, описанному выше.

Доказателсьво.

1. $\Rightarrow$, Если две вершины достижимы друг из друга, то поиск в глубину найдет путь из одной из них в другую, поэтому они окажутся в одном дереве обхода.

2. $\Leftarrow$. Вершина r была рассмотрена вторым обходом в глубину раньше, чем s и t, значит время выхода из нее при первом обходе в глубину больше, чем время выхода из вершин s и t. Из этого мы получаем 2 случая:

   (a) Обе эти вершины были достижимы из r в инвертированном графе. А это означает взаимную достижимость вершин s и r и взаимную достижимость вершин r и t. А складывая пути мы получаем взаимную достижимость вершин s и t.

   (b) Хотя бы одна не достижима из r в инвертированном графе, например t. Значит и r была не достижима из t в инвертированном графе, так как время выхода r — больше . Значит между этими вершинами нет пути, но последнего быть не может, потому что t была достижима из r по пункту (a).

   Значит, из случая 2.(a) и не существования случая 2.(b) получаем, что вершины s и t взаимно достижимы в обоих графах. Теорема доказана.

Время работы алгоритма. Для того, чтобы инвертировать все ребра в графе, представленном в виде списка потребуется $O(V+E)$ действий. Для матричного представления графа не нужно выполнять никакие действия для его инвертирования. Количество ребер в инвертированном равно количеству ребер в изначальном графе, поэтому поиск в глубину будет работать за $O(V+E)$ Поиск в глубину в исходном графе выполняется за $O(V+E)$. В итоге получаем, что время работы алгоритма $O(V+E)$.

Псевдокод.

```
function dfs1(v):
        color[v] = 1
        for (v, u) in E
            if not visited[u]
                dfs1(G[v][u])
        add v to the end of list ord

    function dfs2(v):
        component[v] = col
        for (v, u) in E
            if (u is not in any component still)
                dfs2(H[v][u])

    function main():
        read arrays G and H
        for u in V
            if not visited[u]
                dfs1(u)
        col = 1
        for (all u in reversed ord[])
            if (u is not in any component still)
                dfs2(u)
                col++
```

15 Мосты и точки сочленения в графе.

15.1 Поиск мостов.

Пусть дан неориентированный граф. Мостом называется такое ребро, удаление которого делает граф несвязным (или, точнее, увеличивает число компонент связности). Требуется найти все мосты в заданном графе.

Опишем алгоритм, основанный на поиске в глубину, и работающий за время O(n+m), где n — количество вершин, m — рёбер в графе.

Алгоритм Запустим обход в глубину из произвольной вершины графа; обозначим её через root.

Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины v. Тогда, если текущее ребро (v, to) таково, что из вершины to и из любого её потомка в дереве обхода в глубину нет обратного ребра в вершину v или какого-либо её предка, то это ребро является мостом. В противном случае оно мостом не является. (В самом деле, мы этим условием проверяем, нет ли другого пути из v в to, кроме как спуск по ребру (v, to) дерева обхода в глубину.)

Теперь осталось научиться проверять этот факт для каждой вершины эффективно. Для этого воспользуемся "временами входа в вершину", вычисляемыми алгоритмом поиска в глубину.

Итак, пусть $tin[v]$ — это время захода поиска в глубину в вершину v. Теперь введём массив $fup[v]$, который и позволит нам отвечать на вышеописанные запросы. Время $fup[v]$ равно минимуму из времени захода в саму вершину $tin[v]$, времён захода в каждую вершину p, являющуюся концом некоторого обратного ребра (v, p), а также из всех значений $fup[to]$ для каждой вершины to, являющейся непосредственным сыном v в дереве поиска:

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all } (v,p) - \text{back edge} \\ fup[to], & \text{for all } (v,to) - \text{tree edge} \end{cases}$$

Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын to, что $fup[to] \le tin[v]$. (Если $fup[to] = tin[v]$, то это означает, что найдётся обратное ребро, приходящее точно в v; если же $fup[to] < tin[v]$, то это означает наличие обратного ребра в какого-либо предка вершины v.)

Таким образом, если для текущего ребра (v, to) (принадлежащего дереву поиска) выполняется $fup[to] > tin[v]$, то это ребро является мостом; в противном случае оно мостом не является.

```
void dfs (int v, int p = -1) {
        used[v] = true;
        tin[v] = fup[v] = timer++;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
```

```
                if (to == p)  continue;
                if (used[to])
                        fup[v] = min (fup[v], tin[to]);
                else {
                        dfs (to, v);
                        fup[v] = min (fup[v], fup[to]);
                        if (fup[to] > tin[v])
                                IS_BRIDGE(v,to);
                }
        }
}
```

15.2 Поиск точек сочленения.

Пусть дан связный неориентированный граф. Точкой сочленения называется такая вершина, удаление которой делает граф несвязным.

Опишем алгоритм, основанный на поиске в глубину, работающий за $O(n+m)$, где n — количество вершин, m — рёбер.

Алгоритм Запустим обход в глубину из произвольной вершины графа; обозначим её через root.

1. Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины $v \neq$ root. Тогда, если текущее ребро (v, to) таково, что из вершины to и из любого её потомка в дереве обхода в глубину нет обратного ребра в какого-либо предка вершины v, то вершина v является точкой сочленения. В противном случае, т.е. если обход в глубину просмотрел все рёбра из вершины v, и не нашёл удовлетворяющего вышеописанным условиям ребра, то вершина v не является точкой сочленения. (В самом деле, мы этим условием проверяем, нет ли другого пути из v в to)

2. Рассмотрим теперь оставшийся случай: $v =$ root. Тогда эта вершина является точкой сочленения тогда и только тогда, когда эта вершина имеет более одного сына в дереве обхода в глубину. (В самом деле, это означает, что, пройдя из root по произвольному ребру, мы не смогли обойти весь граф, откуда сразу следует, что root — точка сочленения).

Теперь осталось научиться проверять этот факт для каждой вершины эффективно. Для этого воспользуемся "временами входа в вершину", вычисляемыми алгоритмом поиска в глубину.

Итак, пусть $tin[v]$ — это время захода поиска в глубину в вершину v. Теперь введём массив $fup[v]$, который и позволит нам отвечать на вышеописанные запросы. Время $fup[v]$ равно минимуму из времени захода в саму вершину $tin[v]$, времён захода в каждую вершину p, являющуюся концом некоторого обратного ребра (v, p), а также из всех значений $fup[to]$ для каждой вершины to, являющейся непосредственным сыном v в дереве поиска:

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all } (v,p) - \text{back edge} \\ fup[to], & \text{for all } (v,to) - \text{tree edge} \end{cases}$$

Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын to, что $fup[to] < tin[v]$.

Таким образом, если для текущего ребра (v, to) (принадлежащего дереву поиска) выполняется $fup[to] \geq tin[v]$, то вершина v является точкой сочленения. Для начальной вершины $v =$ root критерий другой: для этой вершины надо посчитать число непосредственных сыновей в дереве обхода в глубину.

```
void dfs (int v, int p = -1) {
        used[v] = true;
        tin[v] = fup[v] = timer++;
        int children = 0;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (to == p)  continue;
                if (used[to])
                        fup[v] = min (fup[v], tin[to]);
                else {
                        dfs (to, v);
                        fup[v] = min (fup[v], fup[to]);
                        if (fup[to] >= tin[v] && p != -1)
                                IS_CUTPOINT(v);
                        ++children;
                }
        }
        if (p == -1 && children > 1)
                IS_CUTPOINT(v);
}
```

16 Нахождение подстроки в строке: префикс-функция, алгоритм Кнута-Морриса-Пратта.

16.1 Префикс-функция.

Определим префикс-функцию от строки s в позиции i следующим образом: $\pi(s,i) = \max\limits_{k=0\ldots i} \{k : \ s[0\ldots k-1] = s[i-k+1\ldots i]\}$

Эффективный алгоритм.

1. Значение $\pi[i+1]$ не более чем на единицу превосходит значение $\pi[i]$ для любого i. Действительно, в противном случае, если бы $\pi[i+1] > \pi[i]+1$, то рассмотрим этот суффикс, оканчивающийся в позиции $i+1$ и имеющий длину $\pi[i+1]$ — удалив из него последний символ, мы получим суффикс, оканчивающийся в позиции i и имеющий длину $\pi[i+1]-1$, что лучше $\pi[i]$, т.е. пришли к противоречию.

2. Избавимся от явных сравнений подстрок. Итак, пусть мы вычислили значение префикс-функции $\pi[i]$ для некоторого i. Теперь, если $s[i+1] = s[\pi[i]]$, то мы можем с уверенностью сказать, что $\pi[i+1] = \pi[i] + 1$. Пусть теперь, наоборот, оказалось, что $s[i+1] \neq s[\pi[i]]$. Тогда нам надо попытаться попробовать подстроку меньшей длины. В целях оптимизации хотелось бы сразу перейти к такой (наибольшей) длине $j < \pi[i]$, что по-прежнему выполняется префикс-свойство в позиции i, т.е. $s[0 \dots j-1] = s[i-j+1 \dots i]$. Действительно, когда мы найдём такую длину j, то нам будет снова достаточно сравнить символы $s[i+1]$ и $s[j]$ — если они совпадут, то можно утверждать, что $\pi[i+1] = j+1$. Иначе нам надо будет снова найти меньшее (следующее по величине) значение j, для которого выполняется префикс-свойство, и так далее. Может случиться, что такие значения j кончатся — это происходит, когда $j = 0$. В этом случае, если $s[i+1] = s[0]$, то $\pi[i+1] = 1$, иначе $\pi[i+1] = 0$. Итак, общая схема алгоритма у нас уже есть, нерешённым остался только вопрос об эффективном нахождении таких длин j. Поставим этот вопрос формально: по текущей длине j и позиции i (для которых выполняется префикс-свойство, т.е. $s[0 \dots j-1] = s[i-j+1 \dots i]$) требуется найти наибольшее $k < j$, для которого по-прежнему выполняется префикс-свойство. После столь подробного описания уже практически напрашивается, что это значение k есть не что иное, как значение префикс-функции $\pi[j-1]$, которое уже было вычислено нами ранее (вычитание единицы появляется из-за 0-индексации строк). Таким образом, находить эти длины k мы можем за $O(1)$ каждую.

```
vector<int> prefix_function (string s) {
        int n = (int) s.length();
        vector<int> pi (n);
        for (int i=1; i<n; ++i) {
                int j = pi[i-1];
                while (j > 0 && s[i] != s[j])
                        j = pi[j-1];
                if (s[i] == s[j])  ++j;
                pi[i] = j;
        }
        return pi;
}
```

Время работы. Время работы алгоритма составит $O(n)$. Для доказательства этого нужно заметить, что итоговое количество итераций цикла while определяет асимптотику алгоритма. Теперь стоит отметить, что j увеличивается на каждом шаге не более чем на единицу, значит максимально возможное значение $j = n-1$. Поскольку внутри цикла while значение j лишь уменьшается, получается, что j не может суммарно уменьшиться больше, чем $n-1$ раз. Значит цикл while в итоге выполнится не более n раз, что дает итоговую оценку времени алгоритма $O(n)$.

16.2 Поиск подстроки в строке. Алгоритм Кнута-Морриса-Пратта

Дан текст t и строка s, требуется найти и вывести позиции всех вхождений строки s в текст t.

Обозначим для удобства через n длину строки s, а через m — длину текста t.

Образуем строку $s + \# + t$, где символ $\#$ — это разделитель, который не должен нигде более встречаться. Посчитаем для этой строки префикс-функцию. Теперь рассмотрим её значения, кроме первых $n+1$ (которые, как видно, относятся к строке s и разделителю). По определению, значение $\pi[i]$ показывает наидлиннейшую длину подстроки, оканчивающейся в позиции i и совпадающего с префиксом. Но в нашем случае это $\pi[i]$ — фактически длина наибольшего блока совпадения со строкой s и оканчивающегося в позиции i. Больше, чем n, эта длина быть не может — за счёт разделителя. А вот равенство $\pi[i] = n$ (там, где оно достигается), означает, что в позиции i оканчивается искомое вхождение строки s (только не надо забывать, что все позиции отсчитываются в склеенной строке $s + \# + t$).

Таким образом, если в какой-то позиции i оказалось $\pi[i] = n$, то в позиции $i - (n+1) - n + 1 = i - 2n$ строки t начинается очередное вхождение строки s в строку t.

Как уже упоминалось при описании алгоритма вычисления префикс-функции, если известно, что значения префикс-функции не будут превышать некоторой величины, то достаточно хранить не всю строку и префикс-функцию, а только её начало. В нашем случае это означает, что нужно хранить в памяти лишь строку $s + \#$ и значение префикс-функции на ней, а потом уже считывать по одному символу строку t и пересчитывать текущее значение префикс-функции.

Итак, алгоритм Кнута-Морриса-Пратта решает эту задачу за $O(n+m)$ времени и $O(n)$ памяти.

17 Стандартные контейнеры: `vector`, `deque`, `queue`, `priority_queue`, `set`, `map`, итераторы, компараторы.

17.1 Контейнеры.

Контейнеры — основа программировнаия. Каждый популярный язык имеет свои фундаментальные контейнеры, такие как массивы или списки. Современные языки программирования обычно включаюти и более сложные контейнеры, например, деревья.

В стандартной библиотеке C++ есть базовые контейнеры и, что даже важнее, обобщенные алгоритмы, например, поиск или сортировка. Эти алгоритмы работают с итераторами — абстракциями над указателями — и применяются к контейнерам или другим последовательностям.

Основная цель контейнеров — хранить множество объектов в едином объекте. Разные контейнеры имеют разные характеристики, такие как скорость, размер и простота использования. Выбор контейнера зависит от характеристик и желаемого поведения.

В C++ контейнеры реализованы с помощью шаблонов классов, поэтому в них можно положить любые объекты. На самом деле не совсем: объекты должны удовлетворять семантике значения, что означает возможность копирования и присваивания. Также некоторые контейнеры требуют сравнений на равенство. Бывают и другие ограничения.

Стандартные контейнеры делятся на две категории: последовательные и ассоциативные контейнеры. Последовательные хранят порядок, с которым новый элемент был добавлен в контейнер. Ассоциативный хранит объекты в возврастающем порядке (отношение может задать пользователь), чтобы увелчить скорость поиска. Например, `deque`, `list`, `vector` — последовательные, а `map`, `multimap`, `set`, `multiset` — ассоциативные.

В дополнение к стандартным контейнерам стандартная библиотека имеет несколько адаптеров для контейнеров, то есть шаблонов классов, которые используют контейнер для предоставления интерфейса. Например, `priority_queue`, `queue` и `stack`.

Кроме того, стандартная библиотека имеет несколкьо так называемых псевдо-контейнеров, похожих на стандартные, но не удовлетворяющие одному или более требованию к стандартными контейнерам. Например, bitset, valarray (нет итераторов), `basic_string`, `string`, `wstring` (нет методов `front` и `back`), `vector<bool>` (специализация шаблона, но в ней нельзя получить указатель на элемент).

17.1.1 `std::vector`.

Динамический массив — последовательный контейнер, похожий на массив, но способный расти при необходимости. Объекты быстро добавляются или удаляются только с конца. Для других позиция вставка и удаление медленные. Объекты хранятся в непрерывной области памяти. Заголовчный файл — `<vector>`.

17.1.2 `std::deque`.

Дек (double-ended-queue) — последовательный контейнер, поддерживающий быстрые вставки и удаления в начало и в конце контейнера. Вставка или удаление в любую другую позицию медленная, но доступ по индексу быстрый к любому элементу. В реализации объекты хранятся не непрерывными кусками. Определен в заголовочном файле `<deque>`.

17.1.3 `std::queue`.

Очередь — последовательность элементов, которая позволяет добавлять элементы в один конец и удалять с другого конца. Эта структура известна как FIFO (first-in, first-out). Заголовочный файл — `<queue>`.

17.1.4 `std::stack`.

Стек — последовательность элементов, которая позволяет добавлять и удалять элементы только с одного конца. Эта структура известна как LIFO (last-in, first-out). Заголовочный файл — `<stack>`.

17.1.5 `std::list`.

Список — последовательный контейнер, который поддерживают быструю вставку или удаление в любую позицию, но не поддерживает доступ к произвольному элементу массива. Объекты хранятся не последовательно. Заголовочный файл — `<list>`.

17.1.6 `std::priority_queue`.

Очередь с приоритетами организована так, что наибольший элемент всегда первый. Можно добавлять элемент в очередь, получать доступ к первому элементу или удалять первый элемент. Заголовочный файл — `<queue>`.

17.1.7 `std::set`, `std::multiset`.

Множество — ассоциативный контейнер, который хранит ключи в возрастающем порядке. В set ключи должны быть уникальными, но multiset разрешает дублирование ключей. Оба шаблона определены в заголовочном файле `<set>`.

17.1.8 `std::map`, `std::multimap`.

Словарь — ассоциативный контейнер, хранящий пары ключей и ассоциированных с ними значений. Ключи определяют порядок элементов в контейнере. В map они должны быть уникальными, однако есть multimap, разрешающий дублирование ключей. Оба шаблона определены в заголовочном файле `<map>`.

17.2 Требования к контейнерам.

Требования к контейнерам делятся на три категории: member types, member functions и exceptions, причем они могут немного отличаться для ассоциативных и последовательных контейнеров.

17.2.1 Member Types.

Здесь перечислены объявления типов для данного контейнера.

Примеры таких типов: `const_iterator` (тип итератора на константные значения), `const_reference` (константная l-value-ссылка на хранимый объект), `difference_type` (знаковый целочисленный тип для разности итераторов), `iterator` (тип итератора на объект), `reference` (тип ссылки на объект), `size_type` (беззнаковый целочисленный тип, который должен вмещать все неотрицательные значения типа `difference_type`), `value_type` (тип хранимого объекта; как правило, это первый параметр шаблона).

Если контейнер поддерживает двунаправленные итераторы, то также должны быть определены `reverse_iterator` и `const_reverse_itera`

Ассоциативные контейнеры должны определять `key_type`, `compare_type` (функция или функтор для сравнения двух ключей `key_type`) и `value_compare` (для сравнения двух объектов `value_type`).

Опционально контейнеры могут определять также типы `pointer` и `const_pointer` как синонимы соответствующих типов в аллокаторе и `allocator_type` для самого аллокатора.

17.2.2 Member Functions.

Большинство методов имеют константную или линейную сложности относительно количества элементов в контейнере, но некоторые методы ассоциативных контейнеров требуют логарифмическую.

В этих требования перечислены сигнатуры конструкторов (общие, для последовательных, для ассоциативных), деструктор, обязательные методы, опциональные методы, методы для последовательных контейнеров и методы для ассоциативных контейнеров.

17.2.2.1 Конструкторы и деструктор.

Все контейнеры:

- `container()`. Константа.
- `container(allocator_type)`. Константа.
- `container(const container& that)`. Линия.
- $\sim$ `container()`. Линия.

Последовательные контейнеры (дополнительно):

- `container(size_type n, const value_type& x)`. Линия от n.
- `container(size_type n, const value_type& x, allocator_type)`. Линия от n.
- `template<InIter> container(InIter first, InIter last)`. Линия от last − first.
- `template<InIter> container(InIter first, InIter last, allocator_type)`. Линия от last − first.

Ассоциативные контейнеры (дополнительно):

- `container(key_compare compare)`. Константа.
- `container(key_compare compare, allocator_type)`. Константа.
- `template<InIter> container(InIter first, InIter last, key_compare compare)`. Линия.
- `template<InIter> container(InIter first, InIter last, key_compare compare, allocator_type)`. Линия.

17.2.2.2 Обязательные методы.

- `iterator begin()`. Константа.
- `const_iterator begin() const`. Константа.
- `void clear()`. Линия.
- `bool empty() const`. Константа.
- `iterator end()`. Константа.
- `const_iterator end() const`. Константа.
- `iterator erase(iterator p)`. Сложность зависит от контейнера. Только для последовательных контейнеров.
- `void erase(iterator p)`. Амортизированно константа. Только для ассоциативных контейнеров контейнеров.
- `iterator erase(iterator first, iterator last)`. Сложность зависит от контейнера. Только для последовательных контейнеров.
- `void iterator erase(iterator first, iterator last)`. Логарифм плюс last − first. Только для ассоциативных контейнеров.
- `size_type max_size( ) const`. Обычно константа.
- `container& operator=(const container& that)`. Линия относительно суммы size()ов.
- `size_type size() const`. Обычно константа.
- `void swap(const container& that)`. Обычно константа.

Также каждый контейнер должен реализоывать операторы равенства и сравнения, либо как методы, либо предпочтительнее как функции уровня пространства имен (они более гибкие, чем методы, например, компилятор может сделать неявное преобразование типа для левого аргумента).

Также если контейнер поддерживает двунаправленные итераторы, он должен реализовывать методы `rbegin` и `rend`.

17.2.2.3 Необязательные методы.

Эти методы предоставляются контейнерами только в том случае, если имеют константную асимптотику.

- `reference at(size_type n)`.
- `const_reference at(size_type n) const`.
- `reference back()`.
- `const_reference back() const`.
- `reference front()`.
- `const_reference front() const`.
- `reference operator[](size_type n)`.
- `const_reference operator[](size_type n)`.
- `void pop_back()`.
- `void pop_front()`.
- `void push_back(const value_type& x)`.
- `void push_front(const value_type& x)`.

17.2.2.4 Методы последовательных контейнеров.

Асимптотика зависит от контейнера.

- `iterator insert(iterator p, const value_type& x)`.
- `void insert(iterator p, size_type n, const value_type& x)`.
- `template<InIter> void insert(iterator p, InIter first, InIter last)`.

17.2.2.5 Методы ассоциативных контейнеров.

Обозначим через N число элементов в контейнере, через M количество элементво в диапазоне и через C результат функции.

- `size_type count(const key_type& k) const`. $C + \log N$.
- `pair<const_iterator, const_iterator> equal_range(const key_type& k) const`. $\log N$.
- `pair<iterator, iterator> equal_range(const key_type& k)`. $\log N$.
- `size_type erase(const key_type& k)`. $C + \log N$.
- `const_iterator find(const key_type& k) const`. $\log N$.
- `iterator find(const key_type& k)`. $\log N$.
- `iterator insert(const value_type& x)`. $\log N$. Для контейнеров, разрещающих дублирование ключей.
- `pair<iterator, bool> insert(const value_type& x)`. $\log N$. Для контейнеров, запрещающих дублирование ключей.
- `iterator insert(iterator p, const value_type& x)`. $\log N$, но если вставка правильная, то есть элемент вставляется сразу после x, то амортизированно константа.
- `template<InIter> void insert(InIter first, InIter last)`. $M \log (N + M)$, но линия, если диапазон уже отсортирован.
- `key_compare key_comp() const`. Константа.
- `const_iterator lower_bound(const key_type& k) const`. $\log N$.
- `iterator lower_bound(const key_type& k)`. $\log N$.
- `value_compare value_comp() const`. Константа.
- `const_iterator upper_bound(const key_type& k) const`. $\log N$.
- `iterator upper_bound(const key_type& k)`. $\log N$.

17.2.3 Exceptions.

Стандартные контейнеры спроектированы так, чтобы они были надежными при работе с исключениями. Большинство методов не могут бросать исключения, а для тех, которые могут, типы исключений жестко заданы.

Если в контейнер добавляется объект (`insert`, `push_front`, `push_back`) и бросается исключение, то контейнер остается в валидном состоянии, как будто добавления не было.

При вставке более одного значения, разные контейнеры ведут себя по-разному. Например, `list` требует, чтобы либо все элементы были добавлены, либо ни одного. А `map` или `set` добавляет только те элементы, которые были успешно добавлены. Если происходит исключение после добавления нескольких элементов из диапазона, то контейнер сохраняет элементы, которые были добавлены успешно.

Методы erase, `pop_back`, `pop_front` никогда не бросают исключения. Метод swap бросает исключение только в случае, когда в функции `Compare` ассоциативного контейнера при копировании или присваивании объектов было брошено исключение.

17.3 Итераторы.

Итераторы — абстракция над указателями, используемая для доступа к объектам внутри контейнеров и других последовательностей. Обычный указатель может указывать на разные элементы в массиве. Оператор `++` сдвигает указатель к следующему элементу, оператор `*` разыменовывает указатель, возвращая значение из массива. Итераторы обобщают понятие указателя, так что те же самые операции примены к любому контейнеру, даже к деревьям и спискам, причем их семантика сохраняется. Заголовочный файл — `<iterator>`.

17.3.1 Виды итераторов.

Всего есть пять категорий итераторов:

1. Input. Позволяют читать последовательность во время прохода. Имеют оператор `++`, но не имеют `--`. Разыменования возвращают r-value, а не l-value, поэтому изменять элементы нельзя.
2. Output. Позволяют записывать последовательность во время прохода. Имеют оператор `++`, но не имеют `--`. Разыменование возможно только для присваивания в него значения. Нельзя сравнивать.
3. Forward. Позволяет получить однонаправленный доступ к последовательности. Можно ссылать или присваивать объекты сколько угодно раз. Можно использовать вместо Input или Output.
4. Bidirectional. Как и forward-итератор, но есть оператор `--`.
5. Random access. Как и bidirectional-итератор, но есть оператор `[]` для доступа к любому индексу в последовательнсоти. Кроме того, можно прибавлять или вычитать любое число, двигаясь на более чем одну позицию за раз. Вычитание двух итераторов возвращает целое число элементов между ними. Таким образом, random-access-итераторы больше всего соответствуют обычным указателям, и наоборот.

Input-, forward-, bidirectional- и random-access-итераторы могут быть `const_iterator`'ами. Их разыменовывание возвращает константное значение.

17.3.2 Безопасность при работе с итераторами.

Самое важное, что нужно помнить об итераторах, — это что они небезопасны. Как и указатели, итераторы могут указывать на контейнер, который уже уничтожен, или на элемент, который уже удален. Итераторы можно двигать за пределами контейнера, как и указатели. Однако если быть немного осторожным, они становятся безопаснее.

Первый шаг к безопасному использованию итераторов - убедиться, что программа никогда не разыменовывает итератор на конец диапазона. Два итератора могут обозначать диапазон, при этом последний элемемент не включается в диапазон, поэтому второй итератор никогда не должен быть разыменован. Также никогда нельзя разыменовывать указатель после последнего элемента, так как он может быть не валиден.

Но даже без этого валидные итераторы могут стать невалидными и, как следствие, небезопасными, например, если элемент, на который указывает итератор, уничтожен. Поведение в этом случае зависит от контейнера. В целом, правило таково, что контейнеры, основанные на узлах (`list`, `set`, `multiset`, `map`, `multimap`) инвалидируются только при удалении узла. Итераторы на контейнеры, основанные на массивах (`deque`, `vector`) инвалидируются только при реаллокации, которая может произойти после вставки или удаления.

17.3.3 Специальные итераторы.

Итераторы есть не только у контейнеров, но и у других объектов. Например, I/O-итераторы для ввода-вывода и так называемые inserter'ы.

В низкоуровневом представлении потоки ничто иное как последовательность символов. Но о них можно думать и как о последовательности объектов, которые могут бысть считаны оператором » или записаны оператором «. Таким образом, стандартная библиотека включает `istreambuf_iterator`, `ostreambuf_iterator`, `istream_iterator` и `ostream_iterator`.

Другой тип итераторов - inserter'ы, которые позволяют вставлять объект в последовательность. Для этого требуется контейнер и, опционально, итератор для указания позиции, на которую должен быть вставлен новый объект. В стандартной библиотеке есть `back_insert_iterator` и `front_insert_iterator`.

17.3.4 Пользовательские итераторы.

Простейший способ определить свой итератор — унаследовать его от шаблонного класса. Первый шаблонный параметр должен указывать тип итератора, а второй — тип объекта.

17.3.5 Константные итераторы.

Каждый контейнер должен предоставлять как тип итератора, так и тип константного итератора. Константные итераторы возвращают методы контейнера в случае, если вызваны от константного контейнера.

Заметим, что `const_iterator` отличается от `const iterator`. Во втором случае это константный объект типа `iterator`, что означает его неизменность, так что, например, его нельзя сдвинуть на другую позицию. С другой стороны, `const_iterator` — это неконстантный объект типа `const_iterator`, поэтому его можно менять. Ключевая разница в том, что `const_iterator`'ы возвращают r-value-ссылки или константные l-value-ссылки, а обычные итераторы — l-value-ссылки. Стандарт позволяет конвертировать `iterator` в `const_iterator`, но не наоборот.

Другая проблема в том, что `iterator` и `const_iterator` сложно сравнивать. Если компилятор выдает ошибку, то можно попробовать поменять порядок итераторов и равенство на неравенство.

17.3.6 Reverse-итераторы.

Каждый контейнер, поддерживающий двунаправленные итераторы или random-access-итераторы, также предоставляет reverse-итераторы: `reverse_iterator` и `const_reverse_iterator`. Они начинаются в последнем элементе контейнера и заканчиваются перед первым.

Стандартная библиотека позволяют превратить тип обычного итератора в тип обратного итератора, так как `reverse_iterator` — просто обертка над обычным итератором. Адаптируемый итератор должен быть двунаправленным или random-access. Его тип можно узнать, вызвав метод `base()`.

По идее обратные итераторы хороши, но на самом деле двунаправленные итераторы и так могут работать в два направления, поэтому можно реализовать инкремент с помощью декремента адаптируемого итератора.

Обратные итераторы сохраняют те же проблемы с `const_iterator`'ами, а также, например, некоторые методы определены только для конкретных типов итераторов, поэтому неприменимы к обратным. Поэтому имеет смысл использовать функцию `base()`.

В качестве точки вставки передать обратный итератор можно, и не произойдет ничего необычного, но если рассмотреть точку удаления, то удалится элемент на одну позицию от итератора.

17.4 Компараторы.

В заголовочном файле `<function>` определено несколько функторов, то есть классов, имеющих оператор `()`. Чаще всего объекты таких классов используются в стандартных алгоритмах. Рассмотрим те из них, которые реализуют операции сравнения над объектами.

Все операторы имеют вид: `template <typename T> bool comparator<T>::operator()(const T& x, const T& y) const`.

Рассмотрим разные компараторы:

- `equal_to`: $x = y$.
- `greater`: $x > y$.
- `greater_equal`: $x \geqslant y$.
- `less`: $x < y$.
- `less_equal`: $x \leqslant y$.
- `not_equal_to`: $x \neq y$.