

Алгоритмы и структуры данных

Экзамен, 2-й семестр

Contents

1	Медиана и порядковые статистики. Алгоритм с линейным временем работы для медианы	5
2	Сортировка массива: пузырьковая, mergesort, quicksort. Алгоритмы и оценки сложности.	7
3	Списки: односвязный и двусвязный	8
4	Бинарные деревья поиска. Вставка, удаление, оценки сложности.	9
5	Графы. Способы их представления в памяти компьютера. Матрицы смежности, матрицы инцидентности, списки связности, представление на двух массивах (CSR).	11
5.1	Матрица смежности	11
5.2	Матрица инцидентности	12
5.3	Список смежности	13
5.4	CSR (Compressed Sparse Row)	13
6	Обходы графов. Обход в ширину, обход в глубину.	15
7	Алгоритмы поиска кратчайших путей. Алгоритм Дейкстры, алгоритм Беллмана-Форда.	18
7.1	Алгоритм Дейкстры	18
7.1.1	Инициализация	18
7.1.2	Шаг алгоритма	18
7.1.3	Время работы	19
7.2	Доказательство корректности алгоритма Дейкстры	19
7.3	Алгоритм Беллмана — Форда	20
7.3.1	Идея алгоритма	20
7.3.2	Псевдокод	21
7.3.3	Время работы	21
7.4	Доказательство корректности алгоритма Беллмана-Форда . . .	21

8	Остовные деревья. Алгоритмы Прима, Краскала, Борувки.	23
8.1	Алгоритм Краскала	23
8.2	Алгоритм Прима	23
8.3	Алгоритм Борувки	24
9	Эвристики для поиска кратчайших путей, алгоритм A*.	25
10	Потоки в сетях. Максимальный поток и минимальный разрез.	27
11	Хеш-функции. Коллизии. Хеш-таблицы. Хеширование. Фильтр Блюма.	28
12	Предикат поворота. Задача пересечения двух отрезков	31
13	Выпуклые оболочки, алгоритма Джарвиса, Грэхема и Quick-Hull.	33
13.1	Алгоритм Джарвиса	33
13.2	Алгоритм Грэхема	34
13.3	Алгоритм QuickHull	35
14	kD-деревья. Окто- и квадро-деревья.	36
15	Многочлены. Метод Горнера. Умножение Карацубы.	38
1	Основная теорема для рекуррентных соотношений. Схема доказательства.	39
2	АВЛ-деревья. Повороты, балансировка.	41
3	Красно-черные деревья. Балансировка (схема).	43
3.1	Операции	44
3.1.1	Вставка	44
3.1.2	Удаление	44
4	Бинарные кучи. Реализация с указателями и на массиве. Добавление и удаление элемента в бинарную кучу.	47
4.1	Восстановление свойств кучи	47
4.1.1	siftDown	47
4.1.2	siftUp	48
4.2	Извлечение минимального элемента	48
4.3	Добавление нового элемента	49

5	Очереди с приоритетами. Наивная реализация, реализация на бинарной куче	51
5.1	Неотсортированный список	51
5.2	Отсортированный массив:	52
5.3	Реализация на бинарной куче	52
6	Потоки в сетях. Задача о максимальном потоке. Алгоритм Форда – Фалкерсона.	53
7	Амортизационный анализ: групповой анализ, банковский метод. Амортизационный анализ для бинарного счетчика	56
7.1	Групповой анализ	56
7.1.1	Увеличение показаний бинарного счетчика	56
7.2	Метод бухгалтерского учёта	58
7.2.1	Увеличение показаний бинарного счетчика	59
8	Амортизационный анализ: метод потенциалов для динамического массива.	60
8.0.1	Расширение динамической таблицы	60
9	Алгоритм Рабина — Карпа, алгоритм Кнута — Морриса — Пратта. Структура данных «бор», алгоритм Ахо — Корасик	64
9.1	Алгоритм Рабина — Карпа	64
9.1.1	Алгоритм	65
9.1.2	Псевдокод	65
9.1.3	Время работы	66
9.2	Алгоритм Кнута — Морриса — Пратта	66
9.2.1	Алгоритм	66
9.2.2	Псевдокод	66
9.2.3	Время работы	67
9.2.4	Оценка по памяти	67
9.3	Структура данных «бор»	67
9.3.1	Пример	67
9.3.2	Построение	67
9.4	Алгоритм Ахо — Корасик.	68
9.4.1	Шаг 1. Построение бора	68
9.4.2	Шаг 2. Преобразование бора	68
9.4.3	Пример автомата Ахо-Корасик	69
9.4.4	Шаг 3. Построение сжатых суффиксных ссылок	70
9.4.5	Использование автомата	70

10 Алгоритм Бойера-Мура. Эвристики стоп-символа и хорошего суффикса.	71
10.1 Эвристика хорошего суффикса	71
10.2 Эвристика стоп-символа	72
11 Расстояние Левенштейна, алгоритм Вагнера — Фишера	74
11.1 Алгоритм Вагнера — Фишера	76
12 Сканирующая прямая. Алгоритм Бентли-Оттоманна для поиска пересечения отрезков	78
12.1 Наивный алгоритм	78
12.2 Алгоритм Бентли — Оттмана	78
13 Диаграммы Вороного. Алгоритм Форчуна.	81
13.1 Алгоритм Форчуна	84
14 Триангуляция Делоне, связь с диаграммами Вороного. Алгоритм построения	93
15 Сумма Минковского. Задача планирования движения робота в среде с препятствиями. Граф видимости.	96
15.1 Граф видимости	96
15.2 Планирование движения	97
15.3 Сумма Минковского	97
15.3.1 Псевдокод	98
15.3.2 Случай невыпуклых фигур	99
16 Отсечение невидимых поверхностей. Z-буфер и алгоритм художника.	100
16.1 Алгоритм z-буфера (z-buffer algorithm)	100
16.2 BSP-деревья	102

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

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

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

Алгоритм с линейным временем работы Идея алгоритма напоминает QuickSort.

Сам алгоритм:

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

Доказательство оценки сложности:

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

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

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

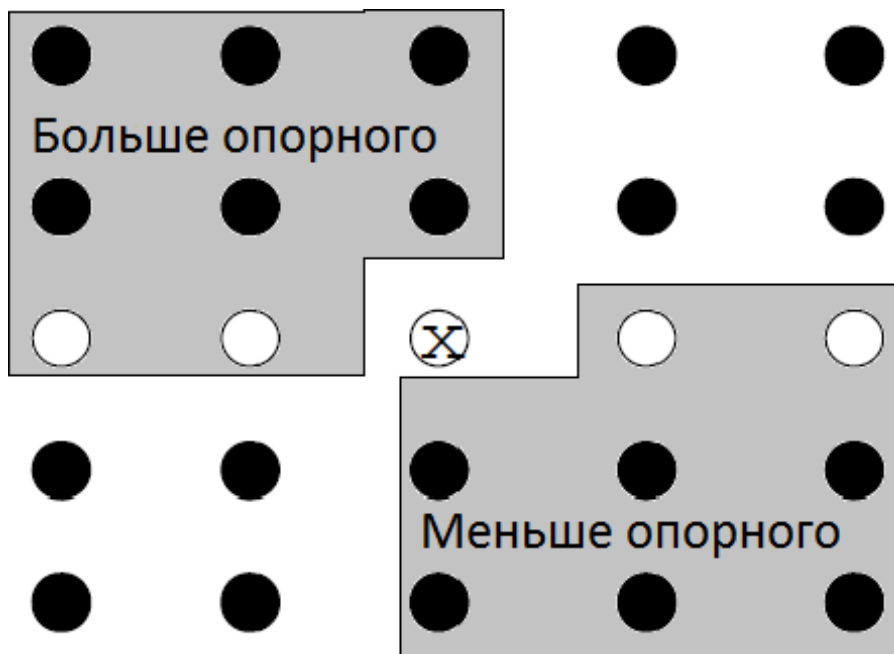


Иллюстрация к рассуждению выше

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

Итоговое время работы:

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

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

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

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

Значит, итоговая сложность алгоритма $O(n)$

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

Пузырьковая сортировка:

Сложность: $O(n^2)$

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

Mergesort (слияние)

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

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

Quicksort

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

Алгоритм:

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

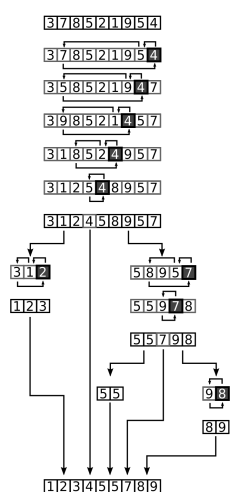


Figure 2: Quicksort

3 Списки: односвязный и двусвязный

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

Бинарное дерево поиска

Структура данных, где каждый отдельно взятый элемент дерева состоит из данных, указателя на левого потомка, указателя на правого потомка, и (опционально) указателя на родителя.

Свойство: Для узла x все узлы в левом поддереве меньше x , а в правом поддереве больше x .

- Нет потомков — лист.
- Нет родителя — корень.

Обход дерева поиска:

- Обход узлов в отсортированном порядке.
- Обход узлов в *прямом* порядке: вершина, левое поддерево, правое поддерево.
- Обход узлов в *обратном* порядке: левое поддерево, правое поддерево, вершина.

Поиск элемента:

В дереве T ищем ключ k . Проверить, есть ли узел с ключом k , вернуть ссылку на узел. **Алгоритм:**

- Если дерево пусто, сообщить, что узел не найден, и стоп.
- Иначе сравнить k со значением ключа корневого узла x .
- Если $k = x$, выдать ссылку на этот узел, остановиться.
- Если $k > x$, рекурсивно искать ключ k в правом поддереве.
- Если $k < x$, рекурсивно искать ключ k в левом поддереве.

Сложность:

- Если дерево сбалансировано: $O(\log n)$.
- Если дерево без развилок (бамбук): $O(n)$, где n — высота дерева.

Добавление элемента:

- Если дерево пусто, замещаем его новым узлом (наш элемент) и останавливаемся.
- Иначе сравниваем k с ключом корневого узла x .
- Если $k = x$, заменить текущий узел новым значением.
- Если $k < x$, рекурсивно добавить в левое поддерево.
- Если $k > x$, рекурсивно добавить в правое поддерево.

Сложность: $O(h)$, где h — высота дерева.

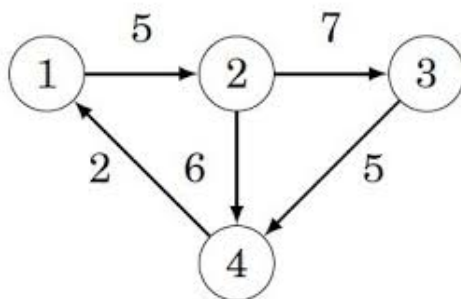
Удаление элемента:

- Если дерево пусто, остановиться.
- Иначе сравнить k с ключом корневого узла x .
- Если $k > x$, рекурсивно удалить из правого поддерева.
- Если $k < x$, рекурсивно удалить k из левого поддерева.
- Если $k = x$, рассмотрим 3 случая:
 1. Нет обоих потомков: удаляем текущий узел и обнуляем ссылку на него.
 2. Есть один потомок: перекидываем с него у родительского узла ссылку на потомка, ссылку на родителя потомка перекидываем на его родителя. Удаляем узел.
 3. Есть оба потомка:
 - Если левый узел правого поддерева отсутствует: копируем из правого узла в удаляемый поле (k, U) .
 - Иначе: возьмём самый левый узел из правого поддерева. Скопируем данные из него в удаляемый. Рекурсивно удалим его.

Сложность: $O(h)$, где h — высота дерева.

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

На протяжении всего билета будем рассматривать граф на иллюстрации ниже



На примере этого взвешенного графа в билете мы будем строить разные представления

Матрицы смежности, матрицы инцидентности, списки связности, представление на двух массивах (CSR).

Опр. 5.1. Графом называется пара $G = (V, E)$, где V — множество вершин графа; $E \subset V \times V$ — множество рёбер графа.

5.1 Матрица смежности

Память: $\Theta(V \times V)$. Элемент на пересечении i строки и j столбца говорит о наличии/отсутствии ребра (i, j) .

Для неориентированного невзвешенного графа:

$A[i][j] = A[j][i] = 1$, если $\exists e = (v_i, v_j)$, иначе $A[i][j] = 0$. A — симметричная матрица.

Для ориентированного невзвешенного графа:

$A[i][j] = 1$, если $\exists e = (v_i, v_j)$, иначе $A[i][j] = 0$. A (в общем случае) несимметричная матрица.

Для ориентированного взвешенного графа:

$A[i][j] = P$, где P — вес ребра $e = (v_i, v_j)$, если $\exists e = (v_i, v_j)$. $A[i][j] = C$, иначе, где C — метка отсутствия ребра, со значением не из диапазона возможных весов.

Временные сложности (Матрица смежности):

1. Проверка смежности вершин x и y : $O(1)$.
2. Перечисление всех вершин, смежных с x : $O(V)$.
3. Определение веса ребра (x, y) : $O(1)$.
4. Подсчёт всех рёбер, инцидентных вершине: $O(V)$.

Для графа из примера:

$$A = \begin{pmatrix} 0 & 5 & 0 & 0 \\ 0 & 0 & 7 & 6 \\ 0 & 0 & 0 & 5 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

5.2 Матрица инцидентности

Граф задаётся двумерным массивом размера $V \times E$, где каждый элемент $I[i][j]$ говорит о существовании (отсутствии) ребра e_j , одним из концов которого является вершина i .

Для неориентированного невзвешенного графа:

$I[i][j] = 1$, если $e_j = (v_i, v)$ или (v, v_i) . $I[i][j] = 0$, иначе.

Для ориентированного невзвешенного графа:

$I[i][j] = 1$, если $e_j = (v, v_i)$ (входит в v_i). $I[i][j] = -1$, если $e_j = (v_i, v)$ (исходит из v_i). $I[i][j] = 0$, иначе.

Для ориентированного взвешенного графа:

Вместо 1/(-1) храним вес. Можно добавить +1-ю строку, где будет храниться вес ребра.

Временные сложности (Матрица инцидентности):

1. Поиск i -й дуги: $O(V)$.
2. Подсчёт кол-ва рёбер, инцидентных вершине x : $O(E)$.
3. Для операций с двумя вершинами: $O(V \cdot E)$.

Для графа из примера

$$A = \begin{pmatrix} -1 & 0 & 0 & 0 & 1 \\ 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 11 & -1 \\ 0 & 7 & 6 & 5 & 2 \end{pmatrix}$$

где -1 — начало ребра, 1 — конец ребра

5.3 Список смежности

Массив указателей на списки, где каждой вершине i соответствует указатель на начало списка смежных с i вершин. В элементах этого списка может храниться информация о соответствующем ребре. Для неориентированных графов элементы списков встречаются в списках обеих вершин. Порядок элементов в списке произвольный.

Временные сложности (Список смежности):

1. Проверка смежности вершин: $O(1)$.
2. Перечисление всех вершин, смежных с x : $O(V)$.
3. Проверка на наличие ребра: $O(1)$.

Для графа из примера список имеет вид

```
1 : 2
2 : 3 4
3 : 4
4 : 1
```

5.4 CSR (Compressed Sparse Row)

Разреженным называется массив, содержащий большое число нулей. CSR — способ хранения разреженной матрицы смежности в виде двух (трёх для взвешенных) массивов.

Алгоритм:

У нас есть разреженная матрица смежности.

1. Составим массив строчных индексов: будем хранить только индексы, с которых начинается очередная последовательность совпадающих цифр от 0 до $(V - 1)$. Получим массив индексов начала информации о вершинах, смежных с той.

2. Массив 'Весы (values)':

5 7 6 5 2 // веса рёбер

3. Массив 'Столбцовые индексы (columns)':

0 1 3 4 // индексация с нуля

4. Массив 'Строчные индексы (rows)':

2 3 4 4 1 // рёбра

Итого:

Граф задаётся 2 (3 для взвешенных) массивами:

1. Содержит индексы столбцов, содержащих ненулевые элементы (кол-во рёбер = E).
2. Показывает индексы массива, определяющие начало и конец каждой строки (кол-во вершин = V).
3. Показывает веса ненулевых рёбер (кол-во рёбер = E).

Оценка памяти:

$O(V + 2E) = O(V + E)$ памяти.

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

Обход графа

Обход графа — способ однократного просмотра всех вершин связного графа в определённом порядке. Обход всегда начинается с заданной вершины. Важно, чтобы каждая вершина была просмотрена и притом только один раз.

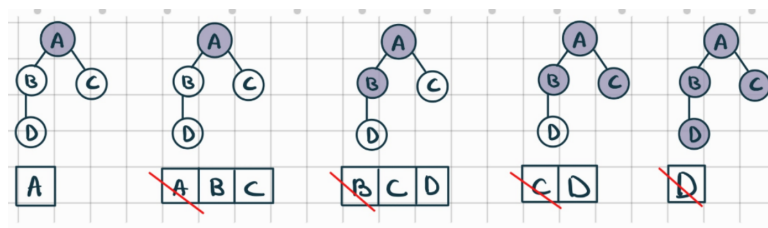
Обход в ширину (BFS)

Пусть задан граф $G = (V, E)$, выделена исходная вершина s . Алгоритм обходит все рёбра для "открытия" всех вершин, достижимых из s , вычисляя при этом расстояние от s до каждой открытой вершины.

- Изначально все вершины белые.
- Когда вершина впервые открыта, красим её в серый.
- Если $(u, v) \in E$ и, если u чёрная, то v либо чёрная, либо серая.
- Все вершины, смежные с чёрной, уже открыты.
- Серые вершины могут иметь белых соседей.
- В процессе обхода строится дерево поиска в ширину с корнем s , содержащее все посещённые вершины.

Алгоритм:

1. Из графа выбирается первая вершина и помечается, как посещённая, заносясь в очередь.
2. Посещается одна вершина из начала очереди, если она не помечена как посещённая. Все её соседние непосещённые вершины заносятся в очередь и в дерево, а сама она из неё удаляется.
3. Повторяется п.2, пока очередь не опустеет.



Пример работы поиска в ширину

Лемма:

В очереди поиска в ширину расстояние от вершины до s монотонно неубывает.

Теорема:

Алгоритм поиска в ширину в невзвешенном графе находит длины кратчайших путей до всех достижимых вершин. (находит в неориентированном графе кратчайший путь по количеству рёбер)

Доказательство:

Допустим, что это не так. Выберем из вершин, для которых кратчайший путь от s найден некорректно, ту, настоящее расстояние до которой \min . Пусть это вершина u , её предок в дереве обхода — вершина v , а предок в кратчайшем пути — вершина w . Т.к. w — предок в кратчайшем пути, то $\rho(s, u) = \rho(s, w) + 1 > \rho(s, w)$. И расстояние до w найдено верно $\rho(s, w) = d[w] \implies \rho(s, u) = d[w] + 1$. Т.к. v — предок в дереве обхода в BFS: $d[u] = d[v] + 1$. Расстояние до u найдено некорректно $\implies \rho(s, u) < d[u]$. $\implies d[w] + 1 < d[v] + 1 \implies d[w] < d[v]$. По лемме в этом случае вершина w попала в очередь и была обработана раньше, чем v . Но w не может быть предком u в дереве обхода в ширину — противоречие.

Анализ времени работы:

В очередь добавляются только непосещённые вершины \implies каждая вершина посещается не более 1 раза. Операции внесения в очередь и удаления — $O(1) \implies$ общее время работы с очередью $O(V)$. Для каждой вершины рассматривается не более $\deg(v)$ рёбер, инцидентных ей. Т. к. $\sum \deg(v) = 2|E| \implies$ время на работу с рёбрами $O(E)$. Общее время работы: $O(V + E)$.

Заметим, что 6.1. $O(V + E)$ получается только при реализации графа на списках смежности и CSR — при представлении графа на матрице смежности оценка $O(V^2)$

Обход в глубину (DFS)

Исследуются все рёбра графа, исходящие из вершины, открытой последней, и покидает её только тогда, когда не осталось неисследованных рёбер. Происходит "откат" в вершину, из которой была открыта вершина v . Процесс повторяется, пока не будут открыты все вершины.

Алгоритм:

1. Выбирается начальная вершина, отмечается как посещённая и заносится в стек.
2. В стеке находится последняя посещённая вершина. Для неё выбирается первая непосещённая вершина и ей присваивается значение "посещение" и добавляется в стек (рекурсивный вызов функции для соседних с последней вершиной в стеке).
3. Если таких вершин нет, то она удаляется из стека, и процесс повторяется для предыдущей вершины.
4. Повторять 1, 2 и 3, пока стек не станет пустым.

Анализ времени работы:

Процедура DFS вызывается от каждой вершины не более 1 раза. Внутри процедуры рассматриваются все такие рёбра $\{e \mid \text{begin}(e) = u\}$. Всего таких рёбер для всех вершин в графе $O(E) \implies$ время работы алгоритма $O(V + E)$.

Заметим, что 6.2. Аналогично поиску в ширину, $O(V + E)$ получается только при реализации графа на списках смежности и CSR — при представлении графа на матрице смежности оценка $O(V^2)$

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

7.1 Алгоритм Дейкстры

Опр. 7.1. Алгоритм Дейкстры — это алгоритм на графах, который находит кратчайшие пути от одной вершины (источника) до всех остальных вершин в графе со взвешенными рёбрами, при условии, что веса всех рёбер неотрицательны.

Входные данные:

- Взвешенный ориентированный или неориентированный граф $G = (V, E)$, где V — множество вершин, E — множество рёбер.
- Функция веса рёбер $w : E \rightarrow \mathbb{R}_{\geq 0}$, где $w(u, v) \geq 0$ для всех $(u, v) \in E$.
- Исходная вершина (источник) $s \in V$.

Выходные данные:

- Для каждой вершины $v \in V$ — длина кратчайшего пути от s до v , $d[v]$.
- Для каждой вершины $v \in V$ (кроме s) — предшествующая вершина на кратчайшем пути от s , $p[v]$.

7.1.1 Инициализация

Для всех вершин $v \in V$: $d[v] = +\infty$, $p[v] = \text{NULL}$. $d[s] = 0$.

Создаём приоритетную очередь PQ , в которую добавляем пары $(d[v], v)$ для всех вершин v . В начале PQ содержит только $(0, s)$.

7.1.2 Шаг алгоритма

Пока PQ не пуста:

1. Извлекаем из PQ вершину u с минимальным значением $d[u]$.
2. Для каждого соседа v вершины u :
 - Если $d[u] + w(u, v) < d[v]$:
 - $d[v] = d[u] + w(u, v)$.
 - $p[v] = u$.
 - Обновляем приоритет v в PQ (или добавляем, если v ещё нет в PQ).

7.1.3 Время работы

Время работы алгоритма Дейкстры зависит от реализации приоритетной очереди Q :

- **Массив:** $O(V^2)$ (для каждой из V вершин осуществляется поиск в массиве за $O(V)$).
- **Бинарная куча:** $O((|V| + |E|) \log |V|) = O(E \log V)$ (в ходе алгоритма каждое ребро из E будет добавлено/удалено из бинарной кучи за $O(\log V)$).
- **Фибоначчиева куча:** $O(|E| + |V| \log |V|)$.

7.2 Доказательство корректности алгоритма Дейкстры

Доказательство корректности алгоритма Дейкстры основывается на **инварианте**, который поддерживается на каждом шаге.

Теорема 7.1 (Корректность алгоритма Дейкстры). Когда алгоритм Дейкстры завершается, для каждой вершины $v \in V$, $d[v]$ равно длине кратчайшего пути от источника s до v .

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

База индукции: На первом шаге выбирается s . Для неё выполнено $d[s] = \delta(s, s) = 0$.

Индукционный шаг: Предположим, что для n первых шагов алгоритм сработал верно и на $n + 1$ шагу выбрана вершина u . Докажем, что в этот момент $d[u] = \delta(s, u)$. Для начала отметим, что для любой вершины v , всегда выполняется $d[v] \geq \delta(s, v)$ (алгоритм не может найти путь короче, чем кратчайший из всех существующих). Пусть P — кратчайший путь из s в u . Пусть z — последняя посещённая вершина на пути P , а v — первая непосещённая вершина на P , следующая за z . Поскольку P кратчайший путь, его часть, ведущая через z в v , тоже кратчайшая, следовательно $\delta(s, v) = \delta(s, z) + w(z, v)$. По предположению индукции, в момент посещения вершины z выполнялось $d[z] = \delta(s, z)$. Следовательно, вершина v получила метку не больше, чем $d[z] + w(z, v) = \delta(s, z) + w(z, v) = \delta(s, v)$. То есть, $d[v] = \delta(s, v)$.

С другой стороны, поскольку сейчас мы выбрали вершину u , её метка минимальна среди непосещённых, то есть $d[u] \leq d[v]$. Комбинируя это с $d[v] = \delta(s, v) \leq \delta(s, u)$ (поскольку P — кратчайший путь и все веса неотрицательны), мы имеем $d[u] \leq \delta(s, u)$. Поскольку, как было сказано ранее, $d[u]$ всегда $\geq \delta(s, u)$, то $d[u] = \delta(s, u)$, что и требовалось доказать.

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

Замечание. Важное условие для корректности алгоритма Дейкстры — это **неотрицательные веса рёбер**. Для графов с отрицательными весами используются другие алгоритмы, такие как Беллмана-Форда.

7.3 Алгоритм Беллмана — Форда

Опр. 7.2. Алгоритм Беллмана-Форда (англ. Bellman-Ford algorithm) — это алгоритм на графах, который находит кратчайшие пути от одной вершины (источника) до всех остальных вершин в графе со взвешенными рёбрами. В отличие от алгоритма Дейкстры, он может работать с графами, содержащими рёбра отрицательного веса, и способен обнаруживать циклы отрицательного веса.

Входные данные:

- Взвешенный ориентированный или неориентированный граф $G = (V, E)$, где V — множество вершин, E — множество рёбер.
- Функция веса рёбер $w : E \rightarrow \mathbb{R}$. Веса могут быть отрицательными.
- Исходная вершина (источник) $s \in V$.

Выходные данные:

- Если граф не содержит циклов отрицательного веса, достижимых из s :
 - Для каждой вершины $v \in V$ — длина кратчайшего пути от s до v , $d[v]$.
 - Для каждой вершины $v \in V$ (кроме s) — предшествующая вершина на кратчайшем пути от s , $p[v]$.
- Если граф содержит цикл отрицательного веса, достижимый из s : возвращает признак наличия такого цикла.

7.3.1 Идея алгоритма

Алгоритм Беллмана-Форда использует принцип динамического программирования. Он итеративно релаксирует все рёбра графа $|V| - 1$ раз. На k -й итерации алгоритм гарантирует, что найдены кратчайшие пути, состоящие максимум из k рёбер. Поскольку кратчайший путь в графе без отрицательных циклов может содержать максимум $|V| - 1$ рёбер, $|V| - 1$ итераций достаточно. Дополнительная $|V|$ -я итерация используется для обнаружения отрицательных циклов.

7.3.2 Псевдокод

Инициализация: Для всех вершин $v \in V$: $d[v] = +\infty$, $p[v] = \text{NULL}$.
 $d[s] = 0$.

Шаг алгоритма

$|V| - 1$ раз выполняем следующие шаги

1. Проходимся по всем рёбрам $(u, v) \in E$.
2. Сравниваем уже найденное расстояние до конечной вершины ребра $d[v]$ с суммой уже найденного расстояния до начальной вершины $d[u]$ и весом текущего рассматриваемого ребра $w(u, v)$.
 - Если новое значение расстояния меньше старого, то обновляем $d[v] = d[u] + w(u, v)$

Псевдокод

```
for i = 1 to |V|-1
  for (u,v) in E
    if d[v] > d[u] + w(u,v) // w(u,v) - вес ребра uv
      d[v] = d[u] + w(u,v)
```

7.3.3 Время работы

Время работы алгоритма Беллмана-Форда: $O(|V| \cdot |E|)$. Это обусловлено тем, что во фазе релаксации выполняется $|V| - 1$ итераций, и на каждой итерации просматриваются все $|E|$ рёбер. Фаза обнаружения также просматривает все $|E|$ рёбер один раз.

7.4 Доказательство корректности алгоритма Беллмана-Форда

Теорема 7.2 (Корректность алгоритма Беллмана-Форда). Если граф G не содержит циклов отрицательного веса, достижимых из s , то после $|V| - 1$ итераций фазы релаксации, $d[v]$ равно длине кратчайшего пути от s до v для всех $v \in V$. Если после $|V| - 1$ итераций возможно дальнейшее улучшение $d[v]$ для какого-либо ребра (u, v) , то в графе существует цикл отрицательного веса, достижимый из s .

Доказательство. Докажем, что после k -й итерации алгоритма, $d[v]$ содержит длину кратчайшего пути от s до v , состоящего максимум из k рёбер, или, по крайней мере, оценивает её сверху.

Лемма 1: Верхняя граница. Для любой вершины v , $d[v]$ всегда $\geq \delta(s, v)$ (где $\delta(s, v)$ — истинная длина кратчайшего пути). Доказательство: Инициализация устанавливает $d[s] = 0 = \delta(s, s)$ и $d[v] = \infty$ для $v \neq s$.

Каждая релаксация $d[v] = d[u] + w(u, v)$ сохраняет это свойство: если $d[u] \geq \delta(s, u)$ и $w(u, v) \geq \delta(u, v)$ (что верно, так как $w(u, v)$ — это вес ребра, а не путь), то $d[u] + w(u, v) \geq \delta(s, u) + \delta(u, v) \geq \delta(s, v)$ (по неравенству треугольника для кратчайших путей). Таким образом, $d[v]$ никогда не становится меньше истинного кратчайшего пути.

Лемма 2: Корректность после k итераций. После k -й итерации фазы релаксации, если существует кратчайший путь от s до v , состоящий из k или менее рёбер, то $d[v]$ равно длине этого кратчайшего пути. Доказательство по индукции по k : **База индукции ($k = 0$):** $d[s] = 0$, что верно для пути из 0 рёбер. **Индукционный шаг:** Предположим, что лемма верна для $k - 1$ итерации. Рассмотрим кратчайший путь $P = s \rightsquigarrow x \rightarrow v$, состоящий из k рёбер. Пусть (x, v) — последнее ребро на этом пути. Тогда часть пути $s \rightsquigarrow x$ состоит из $k - 1$ рёбер и является кратчайшим путём от s до x из $k - 1$ рёбер. По индукционному предположению, после $k - 1$ итераций, $d[x]$ уже было равно $\delta(s, x)$. На k -й итерации, когда мы просматриваем ребро (x, v) , произойдёт релаксация (если ещё не произошло), и $d[v]$ будет обновлено до $d[x] + w(x, v) = \delta(s, x) + w(x, v) = \delta(s, v)$. Таким образом, после k итераций, $d[v]$ будет содержать истинную длину кратчайшего пути длиной до k рёбер.

Завершение доказательства теоремы: По лемме 2 после $|V| - 1$ шага будет получено корректный кратчайший путь длины не более $|V| - 1 \Rightarrow$ мы получили корректный путь (все остальные пути будут непростыми, а значит, в отсутствие отрицательных циклов, не могут уменьшить ответ)

Замечание. Алгоритм Беллмана-Форда более универсален, чем Дейкстра, так как он может обрабатывать отрицательные веса рёбер. Однако его асимптотическая сложность $O(|V| \cdot |E|)$ выше, чем у Дейкстры.

8 Остовные деревья. Алгоритмы Прима, Краскала, Борувки.

Остовное дерево связного графа — это ациклический связный подграф (дерево), в который входят все вершины данного графа.

Пусть T — *ациклическое поддерево* $T \subseteq E$ графа $G = (V, E)$, которое соединяет все вершины G и вес которого минимален. Задача поиска такого T — это задача поиска **минимального остовного дерева**.

Для этого используют алгоритмы **Прима, Краскала и Борувки**. Они жадные (т.е. выбирают вариант, лучший в данный момент).

8.1 Алгоритм Краскала

1. Вначале создаётся пустое множество рёбер
2. Пока возможно, из всех рёбер, добавление которых к имеющемуся множеству не вызовет появления цикла, выбирается ребро минимального веса и добавляется к имеющемуся множеству
3. Когда таких рёбер больше нет, алгоритм завершает свою работу

Исходное множество ребёр можно отсортировать по весу за $O(E \log E)$ (Таким образом, необходимое для добавления ребро можно найти не более, чем за $O(E)$, когда в случае неотсортированного массива поиск минимума затратит $O(E)$, поиск второго также за $O(E)$ и т.д.)

Временная сложность алгоритма Краскала: $O(E \log E)$ (На самом деле $O(E(\log E + \alpha(V)))$), но функция, обратная функции Аккермана на реальных данных сильно ограничена, поэтому мы ей пренебрегаем)

8.2 Алгоритм Прима

1. Берём произвольную вершину, находим ребро, инцидентное данной вершине (с наименьшей стоимостью)
2. Затем рассмотрим рёбра графа, один конец которых принадлежит дереву, а другой нет. Из них выбираем то ребро, стоимость которого наименьшая.
3. Присоединим выбранное ребро к дереву.
4. Рост дерева происходит пока не будут исчерпаны все вершины графа.

Временная сложность алгоритма Прима: $O(E \log V)$ (при реализации на бинарной куче)

$O(V^2)$ при реализации очереди с приоритетом на массиве

8.3 Алгоритм Борувки

1. Каждая вершина G - тривиальное дерево
2. Для каждого дерева T найдём ребро с минимальным весом, инцидентных ему.
3. Добавим все такие рёбра (объединим те деревья, которые соединились общим ребром)
4. Остановимся, когда в графе останется только одно дерево T .

Временная сложность алгоритма Борувки: $O(E \log V)$

9 Эвристики для поиска кратчайших путей, алгоритм A*.

Опр. 9.1. Алгоритм A* (англ. A star) — алгоритм поиска, который находит во взвешенном графе маршрут наименьшей стоимости от начальной вершины до выбранной конечной. Данный алгоритм часто используется при поиске кратчайшего пути на плоскости.

Аналогично алгоритму Дейкстры работает только с графами с неотрицательными весами.

Эвристическая функция

В процессе работы алгоритма для вершин рассчитывается функция $f(v) = g(v) + h(v)$, где

- $g(v)$ — наименьшая стоимость пути в v из стартовой вершины,
- $h(v)$ — эвристическое приближение стоимости пути от v до конечной цели.

Функция $h(v)$ называется **допустимой**, если для любой вершины v значение $h(v)$ меньше или равно весу кратчайшего пути от v до цели.

Так же функция $h(v)$ должна быть **монотонной**

Первое условие является обязательным, при наличии второго условия алгоритм A* является **оптимальным**

Примеры эвристических функций $h(v)$ (будем считать, что вершины графа находятся на плоскости и у каждой есть координаты (x, y))

1. Если возможно перемещение в 4 направлениях, то в качестве эвристики выбирается **манхэттенское расстояние**

$$h(v) = |v.x - goal.x| + |v.y - goal.y|$$

2. **Расстояние Чебышева** применяется, когда к четырем направлениям добавляются диагонали

$$h(v) = \max(|v.x - goal.x|, |v.y - goal.y|)$$

3. Если передвижение не ограничено сеткой, то можно использовать **евклидово расстояние** по прямой

$$h(v) = \sqrt{(v.x - goal.x)^2 + (v.y - goal.y)^2}$$

Сам алгоритм:

Инициализация

Аналогично алгоритму Дейкстры:

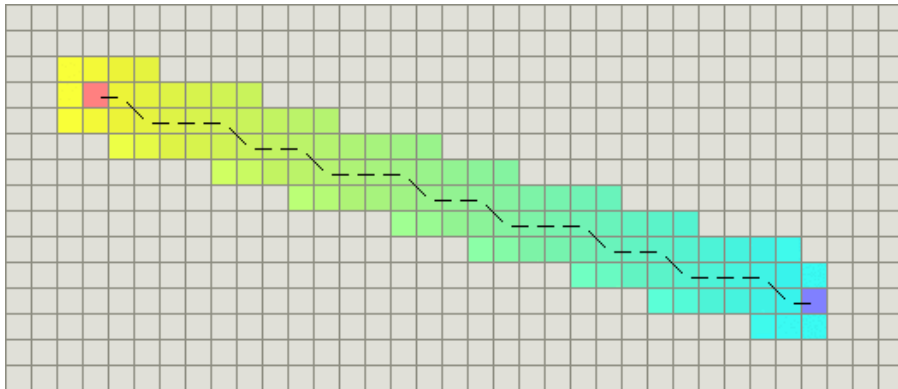
Ищем путь из s в t

Создаём приоритетную очередь PQ , в которую будем добавлять пары $(h(v), v)$ для всех вершин v , где $h(v)$ — эвристическая оценка расстояния от v до t . В начале PQ содержит только $(h(s), s)$. Массив $p[i] = -1$ — для восстановления путей.

Шаг алгоритма

Пока PQ не пуста:

1. Извлекаем из PQ вершину u с минимальным значением $dist[u] + h(u)$ (единственное отличие от алгоритма Дейкстры)
2. Если $u = t$, то все пути найдены, завершаем работу алгоритма.
3. Для каждого соседа v вершины u :
 - Если $dist[u] + w(u, v) < dist[v]$:
 - $dist[v] = dist[u] + w(u, v)$.
 - $p[v] = u$.
 - Добавляем или обновляем $(dist[v] + h(v), v)$ в PQ



Пример A^* на сетке с возможностью ходить в восьми направлениях

Другие эвристики для поиска кратчайших путей

1. **Двунаправленный поиск** — запуск алгоритма A^* из начальной и конечной вершины сразу. Такая эвристика может значительно уменьшить количество посещённых вершин (в случае, если граф является ориентированным, поиск из конечной вершины запускается в противоположных рёбрах)

10 Потоки в сетях. Максимальный поток и минимальный разрез.

См билет 6 из сложных

11 Хеш-функции. Коллизии. Хеш-таблицы. Хеширование. Фильтр Блюма.

Хеш-таблица

Хеш-таблица — это эффективная структура данных для реализации ассоциативных массивов. Используется, когда количество реально хранящихся в массиве ключей мало по сравнению с количеством значений ключей. Она использует массив, размер которого кратен количеству реально хранящихся в нём ключей.

Ассоциативный массив — это абстрактный тип данных, состоящий из пар "ключ-значение". Ключ уникален, для него введена операция сравнения.

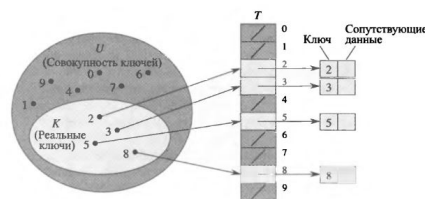


Рис. 11.1. Реализация динамического множества с использованием таблицы с прямой адресацией T . Каждый ключ в совокупности $U = \{0, 1, \dots, 9\}$ соответствует ячейке в таблице. Множество $K = \{2, 3, 5, 8\}$ реальных ключей определяет ячейки в таблице, которые содержат указатели на элементы. Прочие (закрашенные темным цветом) ячейки содержат значение NIL.

Ассоциативный массив

Вместо использования ключа в качестве индекса, хэш-таблица вычисляет индекс по значению ключа. Это позволяет избежать потери памяти.

Коллизии

Таблица с прямой адресацией: элемент ключа k хешируется в ячейку $h(k)$. Величина $h(k)$ называется хеш-значением ключа k .

Принцип Дирихле (в контексте хеш-таблиц) означает ситуацию, называемую коллизией: когда по двум ключам доступна одна и та же ячейка.

Способы разрешения коллизий:

1. **Хеш-таблицы с цепочками:** Каждая ячейка $h(k)$ хранит список (цепочку) элементов, которые хешируются в эту ячейку.

- Операции:

- (a) Вставка: $O(1)$. Вычисляем $h(k)$, вставляем элемент в начало списка, связанного с ячейкой $h(k)$.
- (b) Поиск: Вычисляем $h(k)$, поиск в соответствующем списке.
- (c) Удаление: $O(t)$. Поиск ключа k , удаление в случае успеха.

- Сложность: Если количество элементов в цепочке в среднем одинаково $C_k = n/m$ (где n - количество ключей, m - длина хеш-таблицы), то сложность операций $O(1)$.

2. **Хеш-таблицы с открытой адресацией:** Все элементы хранятся непосредственно в хеш-таблице, без использования связанных списков.

- Для разрешения коллизий: ищем пустую ячейку и добавляем туда.
- Если таблица полностью заполнена, увеличиваем её размер.
- **Стратегии поиска пустой ячейки:**
 - (а) Линейный поиск: просматриваем ячейки $i, i+1, i+2, \dots$ пока не найдём свободную.
 - (б) Квадратичный поиск: смещение не фиксировано, а изменяется квадратично (например, $1, 4, 9, 16, \dots$). Просматриваем $i+1, i+4, i+9, \dots$.

Хеш-функции

Метод деления

$h(k) = k \pmod{m}$. Выбираем m простым, далёким от степени 2.

Пример: $m = 13$

- $K_1 = 1 \rightarrow h(K_1) = 1$
- $K_2 = 100 \rightarrow h(K_2) = 9$
- $K_3 = 12 \rightarrow h(K_3) = 12$
- $K_4 = 179 \rightarrow h(K_4) = 10$
- $K_5 = 2 \rightarrow h(K_5) = 2$
- $K_6 = 444 \rightarrow h(K_6) = 2$

Метод умножения

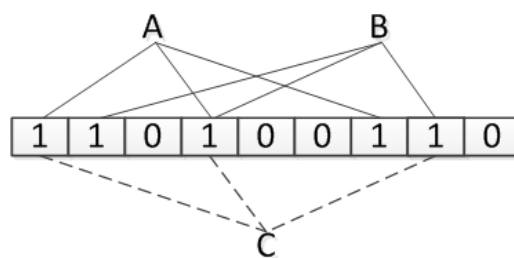
$h(K) = \lfloor m(K \cdot A \pmod{1}) \rfloor$. Желательно, чтобы A не была дробной частью часто числа. $A \in \mathbb{R}, 0 < A < 1$.

Опр. 11.1. Фильтр Блума — вероятностная структура данных позволяющая проверять принадлежность элемента к множеству. При этом существует возможность получить **ложноположительное** срабатывание (элемента в множестве нет, но структура данных сообщает, что он есть), но не **ложноотрицательное**.

То есть данный фильтр даёт два ответа на вопрос о принадлежности элемента к множеству:

1. элемент точно не принадлежит множеству,
2. элемент возможно принадлежит множеству.

Фильтр Блума представляет собой битовый массив из m бит и k различных хеш-функций $h_1 \dots h_k$, равновероятно отображающих элементы исходного множества во множество $\{0, 1, m - 1\}$, соответствующее номерам битов в массиве. Изначально, когда структура данных хранит пустое множество, все m бит обнулены.



Пример фильтра Блума

Для добавления элемента e необходимо записать единицы на каждую из позиций $h_1(e) \dots h_k(e)$ битового массива.

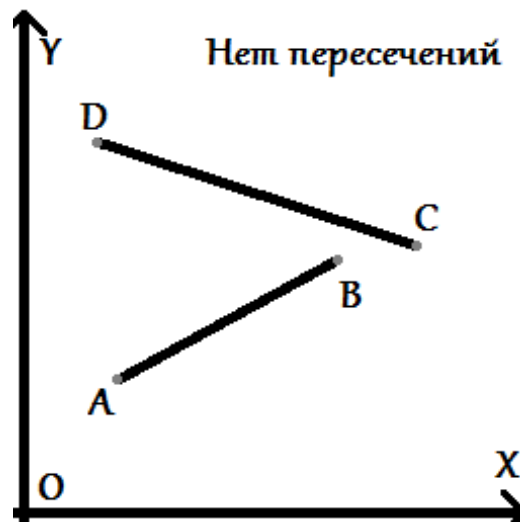
При поиске элемента необходимо проверить наличие единиц на позициях $h_1(e) \dots h_k(e)$ — если хотя бы на одной позиции не будет единицы, элемент точно не присутствует в множестве; п

12 Предикат поворота. Задача пересечения двух отрезков

Даны два отрезка, которые задаются начальной и конечной точками $a, b \in \mathbb{R}^2$ и определяются как множества точек

$$s = \{(1 - t)a + tb, t \in [0, 1]\}.$$

Требуется проверить существование множества их общих точек. Для определения этого факта в вычислительной геометрии используется предикат *левый поворот* (или по часовой стрелке). Рассмотрим возможные расположения точек и самих отрезков относительно друг друга:



Предикат «левый поворот» — выражение

$$\text{LeftTurn}(a, b, c) = \begin{cases} -1 & , \text{ if } (c - a) \times (b - a) < 0 \\ 0 & , \text{ if } (c - a) \times (b - a) = 0 \\ 1 & , \text{ if } (c - a) \times (b - a) > 0 \end{cases}$$

Определить, пересекаются ли два отрезка, можно с помощью предиката поворота. Ясно, что отрезки пересекаются тогда и только тогда, когда для каждого из отрезков его точки не лежат с одной стороны от второго отрезка. Пусть даны отрезки a_0a_1 и b_0b_1 . Отрезки пересекаются, если

$$\begin{aligned} \text{do_intersect} &= \text{orientation}(a_0, a_1, b_0) \neq \text{orientation}(a_0, a_1, b_1) \\ &\text{and } \text{orientation}(b_0, b_1, a_0) \neq \text{orientation}(b_0, b_1, a_1) \end{aligned}$$

В случае, если обе ориентации в одной из строк равны нулю, отрезки лежат на одной прямой, и в этом случае пересечение можно проверить способом, аналогичным пересечению отрезков на действительной прямой (считаем, что точки сравниваются лексикографически):

```
between(x, a0, a1) = (a0 <= x <= a1)
```

```
if a0 > a1  
    swap(a0, a1)
```

```
if b0 > b1  
    swap(b0, b1)
```

```
do_intersect = between(b0, a0, a1) || between(b1, a0, a1) ||  
                between(a0, b0, b1) || between(a1, b0, b1)
```

Если предикат вычисления ориентации был абсолютно точным, то таким же будет описанный алгоритм.

13 Выпуклые оболочки, алгоритма Джарвиса, Грэхема и QuickHull.

Опр. 13.1. Выпуклое множество — такое множество точек, что, для любых двух точек множества, все точки на отрезке между ними тоже принадлежат этому множеству.

Опр. 13.2. Выпуклая оболочка множества точек — такое выпуклое множество точек, что все точки фигуры также лежат в нем.

Опр. 13.3. Минимальная выпуклая оболочка множества точек — это минимальная по площади выпуклая оболочка.

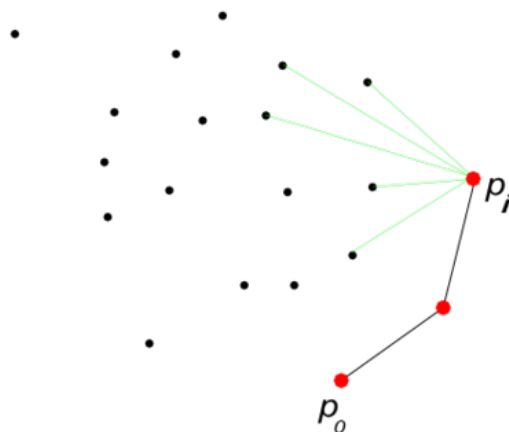
Алгоритмы построения выпуклой оболочки

13.1 Алгоритм Джарвиса

По-другому "Gift wrapping algorithm" (Заворачивание подарка). Он заключается в том, что мы ищем выпуклую оболочку последовательно, против часовой стрелки, начиная с определенной точки.

Описание алгоритма

1. Возьмем точку p_0 нашего множества с самой маленькой координатой (если таких несколько, берем самую правую из них). Добавляем ее в ответ.
2. На каждом следующем шаге для последнего добавленного p_i ищем p_{i+1} среди всех недобавленных точек и p_0 с максимальным полярным углом относительно p_i (Если углы равны, надо сравнивать по расстоянию). Добавляем p_{i+1} в ответ. Если $p_{i+1} == p_0$, заканчиваем алгоритм



Промежуточный шаг алгоритма. Для точки p_i ищем следующую перебором.

Корректность Точка p_0 , очевидно, принадлежит оболочке. На каждом последующем шаге алгоритма мы получаем прямую $p_{i-1}p_i$, по построению которой все точки множества лежат слева от нее. Значит, выпуклая оболочка состоит из p_i -х и только из них.

Сложность

Добавление каждой точки в ответ занимает $O(n)$ времени, всего точек будет k , поэтому итоговая сложность $O(nk)$. В худшем случае, когда оболочка состоит из всех точек сложность $O(n^2)$.

13.2 Алгоритм Грэхема

Алгоритм заключается в том, что мы ищем точки оболочки последовательно, используя стек.

Описание алгоритма

1. Находим точку p_0 нашего множества с самой маленькой y -координатой (если таких несколько, берем самую правую из них), добавляем в ответ.
2. Сортируем все остальные точки по полярному углу относительно p_0 .
3. Добавляем в ответ p_1 — самую первую из отсортированных точек.
4. Берем следующую по счету точку t . Пока t и две последних точки в текущей оболочке p_i и p_{i-1} образуют неправый поворот (вектора $p_i t$ и $p_{i-1} p_i$), удаляем из оболочки p_i .
5. Добавляем в оболочку t .
6. Делаем п.4, пока не закончатся точки

Корректность

Докажем, что на каждом шаге множество P_i -тых является выпуклой оболочкой всех уже рассмотренных точек. Доказательство проведем по индукции.

- **База.** Для трёх первых точек утверждение, очевидно, выполняется.
- **Переход.** Пусть для $i - 1$ точек оболочки совпадают. Докажем, что и для i точек они совпадут.

Рассмотрим истинную оболочку $ch(S \cup i) = ch(S) \cup i \setminus P$, где P — множество всех точек из $ch(S)$, видимых из i . Так как мы добавляли точки в нашу оболочку против часовой стрелки и так как i -тая точка лежит в $ch(S \cup i)$, то P состоит из нескольких подряд идущих последних добавленных в оболочку

точек, и именно их мы удаляем на текущем шаге. Поэтому наша оболочка и истинная для i точек совпадают.

Тогда по индукции оболочки совпадают и для $i = n$.

Сложность

Сортировка точек занимает $O(n \log n)$ времени. При обходе каждая точка добавляется в ответ не более одного раза, поэтому сложность этой части — $O(n)$. Суммарное время — $O(n \log n)$.

13.3 Алгоритм QuickHull

Попытаемся применить к этой задаче подход «Разделяй и властвуй»

Описание Алгоритма

1. Найдем самую левую точку p_0 и самую правую точку p_1 (Если таких несколько, выберем среди таких нижнюю и верхнюю соответственно).
2. Возьмем все точки выше прямой p_0p_1 .
3. Найдем среди этого множества точку p_i , наиболее отдаленную от прямой (если таких несколько, взять самую правую).
4. Рекурсивно повторить шаги 2-3 для прямых p_0p_i и p_ip_1 , пока есть точки.
5. Добавить в ответ точки $p_0 \dots p_i \dots p_1$, полученные в п. 3.
6. Повторить пункты 2-5 для p_1p_0 (то есть для "нижней" половины).
7. Ответ - объединение списков из п. 5 для верхней и нижней половины.

Сложность

Пусть время, необходимое для нахождения оболочки над некой прямой и множеством точек S есть $T(S)$. Тогда $T(S) = O(\|S\|) + T(A \in S) + T(B \in S)$, где A, B — множества над полученными прямыми. Отсюда видно, что в худшем случае, алгоритм тратит $O(n^2)$. На случайных же данных это число равно $O(n \log n)$.

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

KD-деревья

KD-деревья — статическая структура данных для хранения точек в k -мерном пространстве. Позволяет отвечать на запрос, какие точки лежат в данном прямоугольнике.

Построение дерева:

1. Разобьём все точки вертикальной прямой так, чтобы слева (нестрого) и справа (строго) от неё было примерно поровну точек (для этого посчитаем медиану первых координат). Получим подмножества для левого и правого ребёнка.
2. Аналогично строим для этих подмножеств деревья, но разбивать будем уже не вертикальной, а горизонтальной прямой. И т.д.
3. Повторяем 1-2 для каждой половины. (Если $k > 2$, то заменим прямые на гиперплоскость)?

Сложность построения: $T(n) = O(n) + 2T(n/2) \Rightarrow T(n) = O(n \log n)$ - по мастер-теореме. В многомерном случае: $O(n \log^{d-1} n)$, где d - размерность.

Поиск:

Ответить на вопрос, находится ли точка по запросу (количество точек в прямоугольнике). Пусть нам поступил какой-то прямоугольник RR . Нужно вернуть все точки, которые в нём лежат. Будем это делать рекурсивно, получая на вход корень дерева и сам прямоугольник RR . Обозначим область, соответствующую вершине v , как $region(v)$. Она будет прямоугольником, одна или более границ которого могут быть на бесконечности. $region(v)$ можно явно хранить в узлах, записав при построении, или же считать при рекурсивном спуске. Если корень дерева является листом, то просто проверяем одну точку и при необходимости репортим её. Если нет, то смотрим, пересекают ли регионы детей прямоугольник RR . Если да, то запускаемся рекурсивно от такого ребёнка. При этом, если регион полностью содержится в RR , то можно репортить сразу все точки из него. Тем самым мы, очевидно, вернём все нужные точки и только их.

Сложность: $O(\log^2 k)$ в плоском случае в 2D. В многомерном случае: $O(k + \log^d n)$, где k - размер ответа, d - размерность.

Окто- и квадродеревья

Окто-дерево - в $3D$. У узла 8 потомков. **Квадродерево** - в $2D$. У узла 4 потомка.

Построение в 2D (3D аналогично):

1. Возьмём квадрат, в котором находятся все точки.
2. Если всего одна точка, то дерево состоит из 1 листа с этой точкой. Иначе корнем делаем вершину, которая соответствует квадрату.
3. Делим квадрат на 4 равные части и вызываем алгоритм для каждой части.

Глубина квадродерева для множества точек P : $h \sim \log_2 \frac{L}{\varepsilon} + C$, где ε - минимальное расстояние между точками из P , L - сторона исходного квадрата. **Квадродерево содержит** $O(n)$ вершин, где n - всего точек. **Построение:** $O(n \cdot \text{depth})$.

15 Многочлены. Метод Горнера. Умножение Карацубы.

В билете будем рассматривать многочлен $p_n(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ с действительными коэффициентами и $a_i, x \in \mathbb{R}$

Задача: вычислить значение многочлена от конкретного x за наименьшее количество сложных действий (то есть умножений чисел)

Наивный способ: посчитать в лоб (займёт $O(n^2)$ умножений)

Схема Горнера. Постараемся перегруппировать действия, чтобы не было долгих возведений в n -ю степень.

Тогда многочлен $p(x)$ представим в виде:

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \dots x(a_{n-1} + a_nx)) \dots))$$

В данном способе потребуется всего $O(n)$ умножений.

Задача Быстрое умножение чисел на компьютере

Наивная реализация — обычный метод умножения чисел в столбик. В нём каждое число рекуррентно делится на две части и каждая часть числа умножается на каждую часть другого числа. В подсчёте сложности получается такое рекуррентное соотношение

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n)$$

Его решение $T(n) = O(n^2)$

Метод Карацубы Представим наши числа в виде $n_1 = ax + b$, $n_2 = cx + d$ (данная операция осуществляется с помощью битового сдвига) и тогда задача сводится к вычислению коэффициентов многочлена $(ax + b)(cx + d)$ (данное соотношение можно заметить, раскрыв скобки в выражении $(a + b)(c + d)$)

$$\begin{aligned}(ax + b)(cx + d) &= acx^2 + bcx + axd + bd^2 = \\ &= acx^2 + ((a + b)(c + d) - ac - bd)x + bd\end{aligned}$$

В данном способе необходимо выполнить лишь три сложных действия на каждом шаге: умножения $(a + b)(c + d)$, ac и bd

Таким образом получается рекуррентное соотношение:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

Его решение $T(n) = O(n^{\log_2 3})$, что быстрее наивного умножения в столбик.

1 Основная теорема для рекуррентных соотношений. Схема доказательства.

Теорема

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

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

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

Схема доказательства:

Рассмотрим дерево рекурсии данного соотношения. Всего в нем будет $\log_b n$ уровней. На каждом таком уровне, количество детей в дереве будет умножаться на $a > 1$ на уровне i будет a^i потомков. Также известно, что каждый ребенок на уровне i размера $\frac{n}{b^i}$. Ребенок размера $\frac{n}{b^i}$ требует $O\left(\left(\frac{n}{b^i}\right)^d\right)$ дополнительных затрат, поэтому общее количество совершенных действий на уровне i :

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

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

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

Распишем всю работу в течение рекурсивного спуска:

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

Отсюда получаем:

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

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

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

2 АВЛ-деревья. Повороты, балансировка.

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

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

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

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

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

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

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

Малое вращение: ($O(1)$)

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

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

Большое вращение состоит из двух малых.

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

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

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

3 Красно-черные деревья. Балансировка (схема).

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

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

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

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

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

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

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

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

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

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

3.1 Операции

3.1.1 Вставка

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

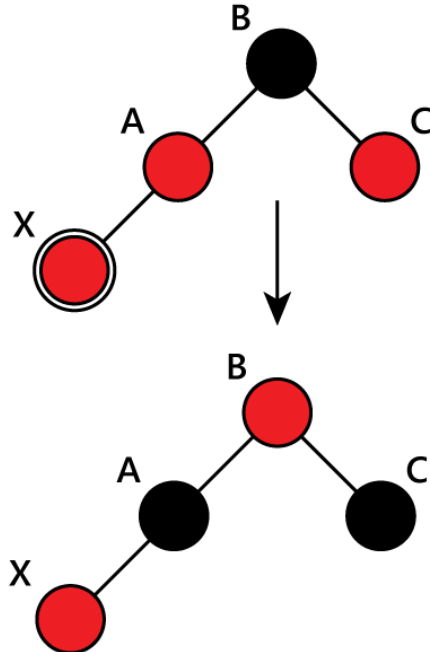
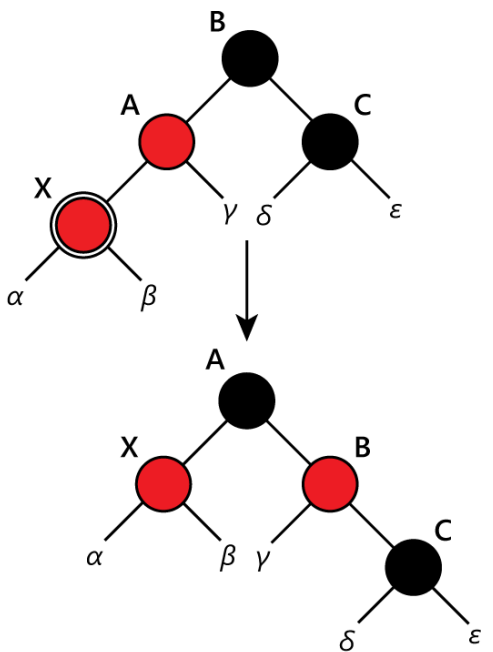
3.1.2 Удаление

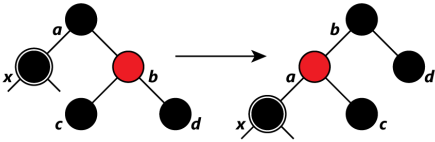
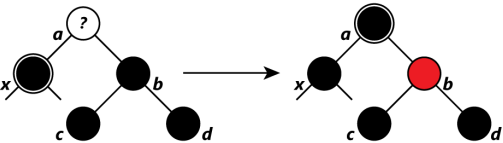
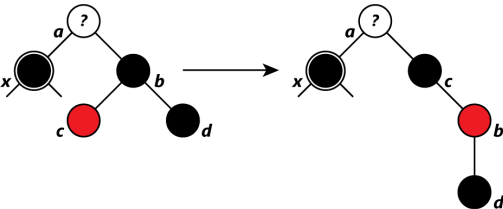
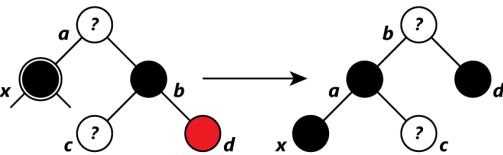
При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

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

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

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

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

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

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

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

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

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

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

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

4.1 Восстановление свойств кучи

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

4.1.1 `siftDown`

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

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

```

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

```

4.1.2 siftUp

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

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

```

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

```

4.2 Извлечение минимального элемента

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

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

```

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

```



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

4.3 Добавление нового элемента

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

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

Реализация на указателях

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

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

Построение кучи

Дан неупорядоченный массив $a[n]$. Требуется построить двоичную кучу.

- `min` в корне (`max` в корне). Делаем нулевой элемент корневым,
• а дальше по очереди добавляем все элементы в конец кучи,
• и каждый раз запускаем `siftUp` (`siftDown`).

Сложность: $O(n \log n)$ — каждый из n элементов за $O(\log n)$.

- Представим, что в массиве хранится дерево ($a[0]$ — корень, потомки $a[i]$ — $a[2i + 1]$ и $a[2i + 2]$). Делаем `siftDown` для вершин, имеющих хотя бы одного потомка: от $\frac{n}{2}$ до 0. (Т.к. поддеревья, состоящие из одной вершины, уже упорядочены). После вызова `siftDown` для вершины её поддеревья являются кучами. Значит, после выполнения всех `siftDown` получим исходную кучу.

Сложность алгоритма: $O(n)$

Доказательство

Число вершин на высоте h в куче из n элементов не превосходит $\lceil \frac{n}{2^h} \rceil$. Высота кучи не превосходит $\log n$. Обозначим за H высоту всего дерева, тогда время построения не превосходит:

$$\sum_{h=1}^H \frac{n}{2^h} \cdot 2h = 2n \sum_{h=1}^H \frac{h}{2^h}$$

Известно, что $\sum_{h=1}^{\infty} \frac{h}{2^h} = 2$. Следовательно, время построения $\leq 2n \cdot 2 = 4n = O(n)$.

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

Очередь с приоритетом

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

Очередь с приоритетом должна поддерживать стандартные операции:

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

Наивная реализация: (не)упорядоченный список/массив

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

5.1 Неотсортированный список

Вставка (`insert`): Сложность: $O(1)$

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

Поиск `min/max`: Сложность: $O(n)$

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

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

5.2 Отсортированный массив:

Вставка (insert): Сложность: $O(n)$

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

Поиск max/min: Сложность: $O(1)$

- Первый/последний элемент списка/массива.

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

5.3 Реализация на бинарной куче

См. билет 6

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

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

Введём необходимые определения

Опр. 6.1. Сеть $G = (V, E)$ — ориентированный граф, в котором:

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

Опр. 6.2. Поток (англ. flow) f в G — действительная функция $f : V \times V \rightarrow \mathbb{R}$, удовлетворяющая условиям:

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

Опр. 6.3. Величина потока f определяется как

$$|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$$

.

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

Дальнейшие определения нам понадобятся для формулировки самого алгоритма

Опр. 6.4. Остаточной пропускной способностью (англ. residual capacity) ребра (u, v) называется величина дополнительного потока, который мы можем направить из u в v , не превысив пропускную способность $c(u, v)$. Иными словами

$$c_f(u, v) = c(u, v) - f(u, v).$$

Опр. 6.5. Для заданной транспортной сети $G = (V, E)$ и потока f , **остаточной сетью** (дополняющая сеть, англ. residual network) в G , порожденной потоком f , является сеть $G_f = (V, E_f)$, где $E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$.

Опр. 6.6. Для заданной транспортной сети $G = (V, E)$ и потока f **дополняющим путем** (англ. augmenting path) p является простой путь из **истока** в **сток** в остаточной сети $G_f = (V, E_f)$.

Опр. 6.7. Определения, которые понадобятся для доказательства корректности алгоритма (s, t) -**разрезом** (англ. $s-t$ cut) $\langle S, T \rangle$ в сети G называется пара множеств S, T , удовлетворяющих условиям:

1. $s \in S, t \in T$
2. $S = V \setminus T$

Опр. 6.8. **Пропускная способность разреза** (англ. capacity of the cut) $\langle S, T \rangle$ обозначается $c(S, T)$ и вычисляется по формуле:

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v).$$

Опр. 6.9. **Поток в разрезе** (англ. flow in the cut) $\langle S, T \rangle$ обозначается $f(S, T)$ и вычисляется по формуле:

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v).$$

Опр. 6.10. **Минимальным разрезом** (англ. minimum cut) называется разрез с минимально возможной пропускной способностью.

Сначала найдём связь между максимальным потоком и потоком через разрез

Теорема 6.1. (Форд — Фалкерсон)

Поток через минимальный разрез равен максимальному потоку.

Доказательство. Сумма потоков из s равна потоку через любой разрез, в том числе минимальный, следовательно, не превышает пропускной способности минимального разреза. Следовательно, максимальный поток не больше пропускной способности минимального разреза.

Осталось доказать, что он и не меньше её. Пускай поток максимален. Тогда в остаточной сети сток не достижим из источника. Пусть A — множество вершин, достижимых из источника в остаточной сети, B — недостижимых. Тогда, поскольку $s \in A, t \in B$, то (A, B) является разрезом. Кроме того, в остаточной сети не существует ребра (a, b) с положительной пропускной способностью, такого что $a \in A, b \in B$, иначе b было бы достижимо из s . Следовательно, в исходной сети поток по любому такому ребру равен его пропускной способности, и, значит, поток через разрез (A, B) равен его пропускной способности. Но поток через любой разрез равен суммарному потоку из источника, который в данном случае равен максимальному потоку. Значит, максимальный поток равен пропускной способности разреза (A, B) , которая не меньше пропускной способности минимального разреза. (ч. т. д)

Сам алгоритм

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

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

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

7.1 Групповой анализ

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

7.1.1 Увеличение показаний бинарного счетчика

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

Increment(A)

```
i = 0
while i < A.length и A[i] == 1
    A[i] = 0
```



```

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

```

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

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

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

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

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

7.2 Метод бухгалтерского учёта

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

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

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

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

7.2.1 Увеличение показаний бинарного счетчика

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

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

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

8 Амортизационный анализ: метод потенциалов для динамического массива.

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

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

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

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

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

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

8.0.1 Расширение динамической таблицы

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

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

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

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

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

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

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

```

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

```

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

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

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

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

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

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

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

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

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

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

9.1 Алгоритм Рабина — Карпа

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

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

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

Рассмотрим хеш $s[0..j]$:

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

Разобьем это выражение на две части:

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

Вынесем из последней скобки множитель p^i :

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

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

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

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

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

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

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

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

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

Получается:

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

9.1.1 Алгоритм

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

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

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

9.1.2 Псевдокод

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

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

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

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

9.1.3 Время работы

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

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

9.2 Алгоритм Кнута — Морриса — Пратта

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

9.2.1 Алгоритм

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

9.2.2 Псевдокод

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

```

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

```

9.2.3 Время работы

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

9.2.4 Оценка по памяти

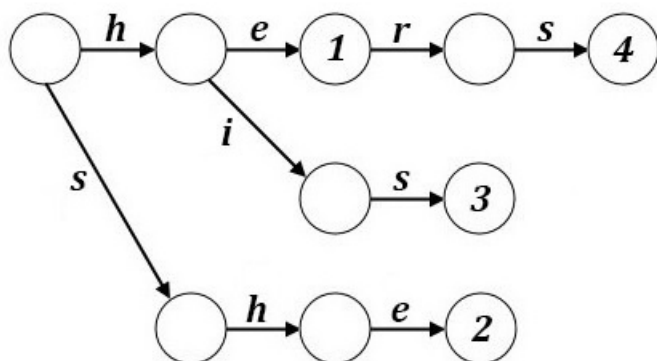
Предложенная реализация имеет оценку по памяти $O(P + T)$. Оценки $O(P)$ можно добиться за счет запоминания значений префикс-функции для позиций в S , меньших $|P| + 1$ (то есть до начала цепочки T). Это возможно, так как значение префикс-функции не может превысить длину образца благодаря разделительному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

9.3 Структура данных «бор»

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

9.3.1 Пример

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



9.3.2 Построение

Введем следующие обозначения:

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

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

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

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

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

9.4 Алгоритм Ахо — Корасик.

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

9.4.1 Шаг 1. Построение бора

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

9.4.2 Шаг 2. Преобразование бора

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

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

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

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

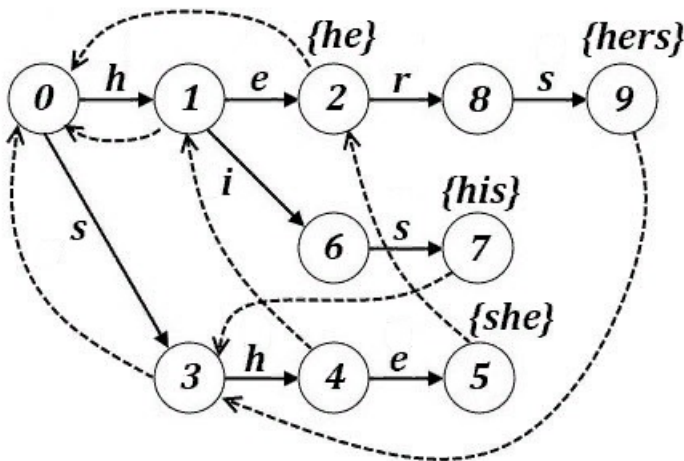
Мы можем понимать рёбра бора как переходы в автомате по соответствующей букве. Однако одними только рёбрами бора нельзя ограничиваться. Если мы пытаемся выполнить переход по какой-либо букве, а соответствующего ребра в боре нет, то мы тем не менее должны перейти в какое-то состояние. Для этого нам и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

Из определений выше можно заметить два следующих факта:

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

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

9.4.3 Пример автомата Ахо-Корасик



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

Суффиксная ссылка для каждой вершины u — это вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине u . Единственный особый случай — корень бора: для удобства суффиксную ссылку из него проведём в себя же. Например, для вершины 5 с соответствующей ей строкой **she** максимальным подходящим суффиксом является строка **he**. Видим, что такая строка заканчивается в вершине 2. Следовательно суффиксной ссылкой вершины для 5 является вершина 2.

9.4.4 Шаг 3. Построение сжатых суффиксных ссылок

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

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

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

9.4.5 Использование автомата

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

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

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

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

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

и попробуем его модернизировать

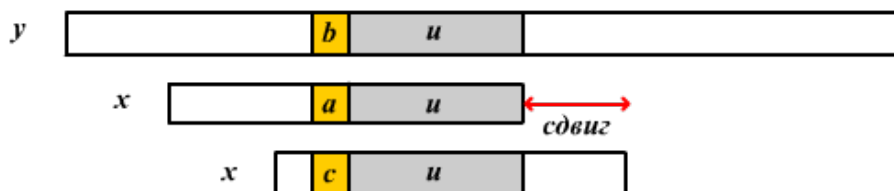
Сам алгоритм: Будем сравнивать все символы в строке с символами подстроки **справа налево**. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и поиск окончен. В случае несовпадения какого-либо символа (или полного совпадения всего шаблона) он использует две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

Будем применять **две эвристики**:

- Эвристика хорошего суффикса.
- Эвристика стоп-символа.

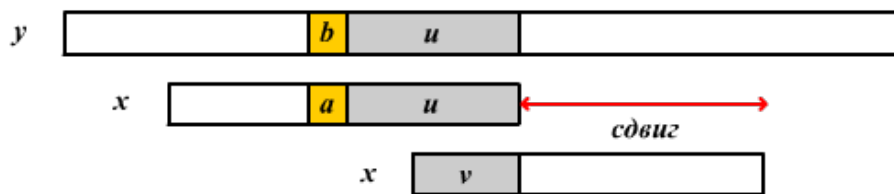
10.1 Эвристика хорошего суффикса

Если при чтении шаблона справа налево совпал суффикс S , а символ b , стоящий перед S в шаблоне (т.е. шаблон имеет вид PbS) не совпал, то эвристика сдвигает шаблон на наименьшее число позиций вправо так, чтобы строка S совпала с шаблоном, а символ, стоящий перед этим совпадением S , отличался бы от b (если такой символ есть).



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

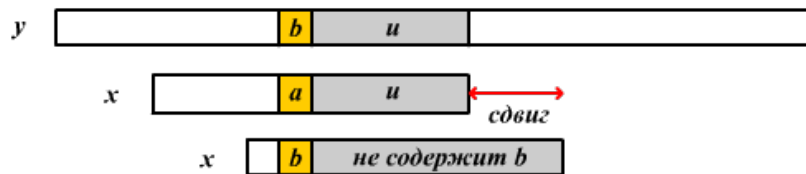
kk



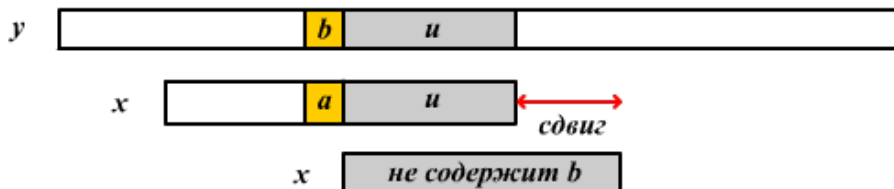
Сдвиг хорошего суффикса, только суффикс подстроки u повторно встречается в x .

10.2 Эвристика стоп-символа

Допустим мы нашли некоторую позицию i , в которой не совпадают символы (пусть в паттерне это символ x , а в исходной строке y , где $x \neq y$). Тогда находим крайнюю правую позицию в паттерне, которая соответствует символу y и сдвигаем паттерн так, чтобы y оказалось под y в исходной строке. Если такой позиции нет — сдвигаем за границу.



Сдвиг плохого символа, символ a входит в x .



Сдвиг плохого символа, символ b не входит в x .

Препроцессинг:

1. Построение таблицы смещений для паттернов

- Таблица содержит список всех символов в паттерне
- В соответствие каждому символу ставится его порядковый номер, считая с конца строки. Если символ встречается несколько раз, то используется самое правое вхождение.

2. Построение таблицы суффиксов:

- Для каждого возможного суффикса t данного шаблона S указываем наименьшую величину, на которую нужно сдвинуть шаблон, чтобы он снова совпал с t и при этом символ, предшествующий этому вхождению t , не совпадал бы с символом, предшествующим суффиксу t .

- Если такой сдвиг невозможен, ставится $|S| = m$.

Доказательство корректности

Для доказательства корректности алгоритма достаточно показать, что если та или иная эвристика предлагает сдвиг более чем на одну позицию вправо, на пропущенных позициях шаблон не найдётся.

Итак, пусть совпал суффикс S , строка-шаблон равна PbS , стоп-символ — a (в дальнейшем малые буквы — символы, большие — строки).

```
Строка:      * * * * * a [-- S --] * * * *
Шаблон:      [--- P ---] b [-- S --]
```

Эвристика стоп-символа. Работает, когда в строке V символ отсутствует. Если $P = WaV$ и в строке V нет символа a , то эвристика стоп-символа предлагает сдвиг на $|V| + 1$ позицию.

```
Строка:      * * * * * a [-- S --] * * * * *
Шаблон:      [- W -] a [--- V ---] b [-- S --]
Пропустить:  [- W -] a [--- V ---] b [-- S --]
Новый шаг:   [- W -] a [--- V ---] b [-- S --]
```

Действительно, если в строке V нет буквы a , нечего пробовать пропущенные $|V|$ позиций. Если же в шаблоне нет символа a , то эвристика стоп-символа предлагает сдвиг на $|P| + 1$ позицию — и также нет смысла пробовать пропущенные $|P|$.

Эвристика совпавшего суффикса. Сама неформальная фраза — «наименьшая величина, на которую нужно сдвинуть вправо шаблон, чтобы он снова совпал с S , но символ перед данным совпадением с S (если такой символ существует) отличался бы от b » — говорит, что меньшие сдвиги бесполезны.

```
Строка:      * * * * * a [-- S --] * * * * *
Шаблон:      [--- P ---] b [-- S --]
Пропустить:  [--- P ---] b [-- S --]
Новый шаг:   [--- P ---] b [-- S --]
```

11 Расстояние Левенштейна, алгоритм Вагнера — Фишера

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

Для расстояния Левенштейна справедливы следующие утверждения:

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

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

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

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

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

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

$d(S_1, S_2) = D(M, N)$, где

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

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

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

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

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

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

11.1 Алгоритм Вагнера — Фишера

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

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

```

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

```

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

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

```

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

```

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

Сканирующая прямая — прямая, проходящая по точкам (отсортированным по какому-то признаку, например, по координате).
Пример: в автомате S_v по координатам.

Характеристические положения

Положение	Действие / Результат
Начало отрезка X	\Rightarrow +2 новых пар соседних отрезков
Конец отрезка	\Rightarrow +1 новая пара отрезков
Точка пересечения отрезков	\Rightarrow +2 пары

Визуализация характерных положений (из исходного документа)

12.1 Наивный алгоритм

Задача: поиск точек пересечения пар отрезков на плоскости.

Входные данные:

- пары начал отрезков
- пары концов отрезков

Выходные данные: точки пересечения с указанием пар отрезков.

Алгоритм: Проверки на пересечение каждого отрезка.

Сложность: $O(n^2)$.

12.2 Алгоритм Бентли — Оттмана

Пусть:

- нет вертикальных отрезков
- пересекается не более чем 11 отрезков (возможно, опечатка, 11 - необычное число, обычно имеется в виду константа)
- Начало (конец) отрезка не является точкой пересечения.

(данные ограничения необязательны, просто при их наличии намного проще формулировать алгоритм — при необходимости каждое ограничение можно убрать)

Усовершенствуем наивный алгоритм.

1. Если отрезки не были соединены (в смысле порядка) по X точек пересечения отрезков, то они не могут пересечься.
2. Характерные положения заметяющей прямой определяют соседние отрезки (события).

Определим события:

- Начало отрезка \implies добавление пары
- Конец отрезка \implies удаление пары
- Пересечение т.ч. \implies поменять местами

В течение некоторого периода остаются постоянными, что приводит к тому, что прямая может двигаться дискретно.

События будем хранить в очереди с приоритетом (PQ), которая движется слева направо (min по координате).

Отрезки будем хранить в бинарном дереве поиска, т.к. необходимо вычислять соседей (BT).

Алгоритм.

Инициализация: $BT = \emptyset$, $Ans = \emptyset$ (структура для ответа).

- PQ заполнена точками начал и концов отрезков.

Шаг: Пока PQ не пусто, извлекаем событие e .

1. e — **начало отрезка** a

- Добавляем a в BT.
- Находим соседей a в дереве BT: t, b . Проверяем на пересечение a, t и a, b .
- Добавляем событие в PQ, и если есть пересечение, то добавляем пары в Ans .

2. e — **конец отрезка** a

- Находим соседей отрезка a в дереве BT: b, t .
- Удаляем a из BT.
- Проверяем пересечение b и t .
- Добавляем событие в PQ, и если есть пересечение, то добавляем пары в Ans .

3. e — **пересечение отрезков** a, b

- Находим верхнего соседа $a(t)$ и нижнего соседа $b(k)$ в ВТ.
- Меняем a и b в ВТ местами.
- Проверяем пересечение пары (t, b) и пары (a, k) .
- Добавляем событие в PQ, и если есть пересечение, то добавляем пары в Ans .

При вставке можно удалять событие (t, b) . При смене порядка можно удалять событие (a, t) .

Сложность алгоритма: $O(n \log n)$

- Сортировка событий: $O(n \log n)$
- Поиск/удаление/вставка в дерево: $O(\log n)$ (в случае событий)
- Проверка на пересечение: $O(\log n)$ (в случае событий)
- Всего событий $n \Rightarrow O(n \log n)$

Итого: $O(n \log n)$

13 Диаграммы Вороного. Алгоритм Форчуна.

Опр. 13.1. Обозначим $dist(p, q)$ евклидово расстояние между двумя точками p и q . На плоскости имеет место формула:

$$dist(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Пусть $P := \{p_1, p_2, \dots, p_n\}$ — множество n различных точек на плоскости. Определим диаграмму Вороного множества P как разбиение плоскости на n ячеек, по одной для каждого центра из P , обладающее тем свойством, что точка q принадлежит ячейке, соответствующей центру p_i , тогда и только тогда, когда $dist(q, p_i) < dist(q, p_j)$ для любой точки $p \in P$ с $i \neq j$. Диаграмму Вороного множества P будем обозначать $Vor(P)$. Допуская некоторую вольность выражений, мы иногда будем употреблять термин « $Vor(P)$ » или «диаграмма Вороного» для обозначения одних лишь ребер и вершин разбиения. Например, говоря, что диаграмма Вороного связна, мы имеем в виду, что объединение ребер и вершин образует связное множество. Ячейка $Vor(P)$, соответствующая центру p_i , обозначается $\mathcal{V}(p_i)$; будем называть ее ячейкой Вороного для p_i .

Теперь присмотримся к диаграмме Вороного повнимательнее. Сначала изучим строение одной ячейки Вороного. Для двух точек p и q на плоскости назовем срединным перпендикуляром p и q перпендикуляр, к отрезку pq , проходящий через его середину. Срединный перпендикуляр делит плоскость на две полуплоскости. Обозначим $h(p, q)$ открытую полуплоскость, содержащую p , а $h(q, p)$ — открытую полуплоскость, содержащую q . Заметим, что $r \in h(p, q)$ тогда и только тогда, когда $dist(r, p) < dist(r, q)$. Отсюда вытекает следующее наблюдение.

Заметим, что 13.1. $V(p_i) = \bigcap_{1 \leq j \leq n, i \neq j} h(p_i, p_j)$.

Таким образом, $V(p_i)$ — пересечение $n - 1$ полуплоскостей, а, значит, является выпуклой многоугольной областью (возможно, неограниченной), имеющей не более $n - 1$ вершин и не более $n - 1$ ребер.

Как выглядит полная диаграмма Вороного? Мы только что видели, что каждая ячейка диаграммы представляет собой пересечение нескольких полуплоскостей, поэтому диаграмма Вороного — это планарное разбиение с прямолинейными ребрами. Одни ребра являются отрезками, другие — полупрямыми. Если только не все центры коллинеарны, то ребер, являющихся полными прямыми, не будет.

Теорема 13.1. Пусть P — множество, содержащее n точек на плоскости (центров). Если все центры коллинеарны, то диаграмма Вороного состоит

из $n - 1$ параллельных прямых. Иначе $Vor(P)$ связна, и ее ребра являются отрезками или полупрямыми.

Доказательство. Первую часть теоремы доказать легко, поэтому предположим, что не все центры коллинеарны. Сначала покажем, что ребра $Vor(P)$ являются либо отрезками, либо полупрямыми. Мы уже знаем, что ребра $Vor(P)$ — части прямых линий, а именно срединных перпендикуляров пары центров. Предположим противное — что ребро e диаграммы $Vor(P)$ является полной прямой. Пусть e лежит на границе ячеек Вороного $V(p_i)$ и $V(p_j)$. Пусть $p_k \in P$ — точка, не лежащая на одной прямой с p_i и p_j . Срединный перпендикуляр p_j и p_k не параллелен e , а, значит, пересекает e . Но тогда часть e , лежащая внутри $h(p_k, p_j)$, не может находиться на границе $V(p_j)$, поскольку она ближе к p_k , чем к p_j . Мы получили противоречие.

Остается доказать, что $Vor(P)$ связна. Если бы это было не так, то существовала бы ячейка Вороного $V(p_i)$, делящая диаграмму на две части. Поскольку ячейки Вороного выпуклы, то $V(p_i)$ состояла бы из полосы, ограниченной двумя параллельными прямыми. Но мы только что доказали, что ребра диаграммы Вороного не могут быть полными прямыми. Противоречие!

Теперь, понимая строение диаграммы Вороного, исследуем ее сложность, т. е. оценим общее число вершин и ребер. Поскольку существует n центров и у каждой ячейки Вороного не более $n - 1$ вершин и ребер, то сложность $Vor(P)$ в худшем случае квадратичная. Не ясно, однако, действительно ли $Vor(P)$ может иметь квадратичную сложность: легко построить пример, когда сложность ячейки Вороного линейна, но может ли оказаться, что у многих ячеек линейная сложность? Следующая теорема показывает, что это не так и что среднее число вершин ячейки Вороного меньше шести.

Теорема 13.2. Для $n > 3$ число вершин в диаграмме Вороного множества n точек на плоскости не превосходит $2n - 5$, а число ребер не больше $3n - 6$.

Доказательство. Если все центры коллинеарны, то теорема сразу следует из предыдущей, поэтому будем предполагать, что это не так. Используем для доказательства формулу Эйлера, согласно которой для любого связного планарного графа с m_v вершинами, m_e ребрами и m_f гранями имеет место следующее соотношение:

$$m_v - m_e + m_f = 2.$$

Мы не можем применить формулу Эйлера непосредственно к $Vor(P)$, потому что $Vor(P)$ содержит полубесконечные ребра и потому не является настоящим графом. Чтобы исправить положение, добавим одну дополнительную вершину v_∞ находящуюся «в бесконечности», и соединим

все полубесконечные ребра $Vor(P)$ с этой вершиной. Теперь мы получили связный планарный граф, к которому можно применить формулу Эйлера. Получаем следующее соотношение между n_v , количеством вершин $Vor(P)$, n_e , количеством ребер $Vor(P)$, и n , количеством граней:

$$(n_v + 1) - n_e + n = 2. \quad (1)$$

Далее, каждое ребро в пополненном графе имеет ровно две вершины, поэтому если просуммировать степени всех вершин, то получится удвоенное количество рёбер. Поскольку степень каждой вершины, включая v_∞ , не меньше трех, получаем:

$$2n_e \geq 3(n_v + 1) \quad (2)$$

В сочетании с формулой (Диаграммы Вороного. Алгоритм Форчуна.) это доказывает теорему.

Мы знаем, что ребра являются частями срединных перпендикуляров пар центров и что вершины — это точки пересечения срединных перпендикуляров. Количество срединных перпендикуляров квадратично зависит от числа центров, тогда как сложность $Vor(P)$ всего лишь линейна. Следовательно, не все срединные перпендикуляры определяют ребра $Vor(P)$, и не все их пересечения являются вершинами $Vor(P)$. Чтобы понять, какие срединные перпендикуляры и точки их пересечения формируют отличительные характеристики диаграммы Вороного, дадим следующее определение. Для точки q определим наибольший пустой круг q относительно P ($C_P(q)$) как наибольший круг с центром в q , который не содержит внутри себя ни одного центра из P . Следующая теорема характеризует вершины и ребра диаграммы Вороного.

Теорема 13.3. Для диаграммы Вороного $Vor(P)$ множества точек P справедливы следующие утверждения:

1. Точка q является вершиной $Vor(P)$ тогда и только тогда, когда граница ее наибольшего пустого круга $C_P(q)$ содержит три или более центров из P .
2. Срединный перпендикуляр центров p_i и p_j определяет ребро $Vor(P)$ тогда и только тогда, когда существует точка q на нем такая, что граница $C_P(q)$ содержит оба центра p_i и p_j и никаких других центров.

Доказательство.

1. Предположим, что существует такая точка q , что граница $C_P(q)$ содержит три или более центров. Пусть p_i , p_j и p_k — три таких центра. Поскольку внутренность $C_P(q)$ пуста, q должна лежать на границе

каждой из ячеек $V(p_i)$, $V(p_j)$ и $V(p_k)$, и q должна быть вершиной $Vor(P)$.

С другой стороны, каждая вершина q диаграммы $Vor(P)$ инцидентна по меньшей мере трем ребрам и, следовательно, по меньшей мере трем ячейкам Вороного $V(p_i)$, $V(p_j)$ и $V(p_k)$. Вершина q должна находиться на равных расстояниях от p_i , p_j и p_k , и не может существовать другого центра, более близкого к q , поскольку в противном случае $V(p_i)$, $V(p_j)$ и $V(p_k)$ не сошлись бы в q . Следовательно, внутренность круга, на границе которого лежат p_i , p_j и p_k , не содержит ни одного центра.

2. Предположим, что существует точка q , обладающая указанным в теореме свойством. Поскольку внутренность круга $C_P(q)$ не содержит ни одного центра, а центры p_i и p_j лежат на его границе, то $dist(q, p_i) = dist(q, p_j) \leq dist(q, p_k)$ для любого $1 \leq k \leq n$. Отсюда следует, что q лежит на ребре или является вершиной $Vor(P)$. Но из первой части теоремы вытекает, что q не может быть вершиной $Vor(P)$. Значит, q лежит на ребре $Vor(P)$, которое определяется срединным перпендикуляром p_i и p_j .

Обратно, пусть срединный перпендикуляр p_i и p_j определяет ребро диаграммы Вороного. На границе наибольшего пустого круга любой точки q во внутренней части этого ребра должны находиться p_i и p_j и ни одного другого центра.

13.1 Алгоритм Форчуна

Принятая в алгоритме стратегия заключается в заметании плоскости прямой, опускающейся сверху вниз. В процессе заметания мы пересчитываем показатели, описывающие вычисляемую структуру. Точнее, пересчитывается информация о пересечении структуры с заметающей прямой. Эта информация изменяется лишь в некоторых точках — *точках событий*.

Попробуем применить эту общую стратегию к вычислению диаграммы Вороного множества $P = \{p_1, p_2, \dots, p_n\}$ центров на плоскости. В соответствии с идеей заметания плоскости мы двигаем горизонтальную заметающую прямую l сверху вниз. При этом мы должны пересчитывать пересечение диаграммы Вороного с заметающей прямой. К сожалению, это не так просто, потому что часть $Vor(P)$ над l зависит не только от центров, лежащих выше l , но и от центров, лежащих ниже l . Иначе говоря, когда заметающая прямая достигает самой верхней вершины ячейки Вороного $V(p_i)$, она еще не встретила соответствующий центр p_i . Следовательно, у нас нет информации, необходимой для вычисления вершины. Мы вынуждены применить идею заметания плоскости немного по-другому: вместо пересчета

пересечения диаграммы Вороного с заметающей прямой, будем хранить информацию о части диаграммы Вороного для центров выше l , которая не может измениться из-за центров ниже l .

Обозначим l^+ замкнутую полуплоскость, расположенную выше l . Какая часть диаграммы Вороного выше l больше не может измениться? Иначе говоря, для каких точек $q \in l^+$ мы точно знаем ближайший к ним центр? Расстояние от точки $q \in l^+$ до любого центра, находящегося под l , больше расстояния от q до самой l . Поэтому ближайший к q центр не может находиться ниже l , если q удалена от какого-то центра $p_i \in l^+$ на расстояние, не большее, чем q от l . Геометрическое место точек, расположенных к некоторому центру $p_i \in l^+$, ближе, чем к l , ограничено параболой. Поэтому геометрическое место точек, расположенных ближе к какому-нибудь центру, находящемуся выше l , чем к самой l , ограничено дугами парабол. Назовем эту последовательность параболических дуг *линией прибоа*. Линию прибоа можно наглядно представить и по-другому. Каждый центр p_i , находящийся выше заметающей прямой, определяет полную параболу β_i . Линия прибоа — это график функции, которая для каждой координаты x принимает значение, совпадающее с самой нижней точкой всех таких парабол.

Заметим, что 13.2. *Линия прибоа x -монотонна, т. е. каждая вертикальная прямая пересекает ее ровно в одной точке.*

Легко видеть, что одна парабола может несколько раз встречаться в линии прибоа. О том, сколько именно раз, мы подумаем позже. А пока заметим, что точки излома на стыке дуг разных парабол, образующих линию прибоа, лежат на ребрах диаграммы Вороного. И это не случайное совпадение: точки излома точно вычерчивают диаграмму Вороного по мере того, как заметающая прямая опускается вниз. Эти свойства линии прибоа легко доказать с помощью элементарных геометрических рассуждений.

Поэтому вместо того чтобы пересчитывать пересечение $Vor(P)$ с l по мере опускания заметающей прямой, мы будем пересчитывать линию прибоа. Мы не станем хранить линию прибоа явно, потому что она непрерывно изменяется при перемещении l . Ненадолго оставим в стороне вопрос о том, как представлять линию прибоа, и сначала разберемся, когда и как изменяется ее комбинаторная структура. Это происходит, когда появляется новая параболическая дуга и когда старая дуга схлопывается в точку и исчезает.

Сначала рассмотрим событие появления новой дуги на линии прибоа. В-первых, это может случиться, когда заметающая прямая l доходит до нового центра. В первый момент парабола, определяемая этим центром, вырождена и имеет нулевую ширину: это вертикальный отрезок, соединяющий новый центр с линией прибоа. По мере того как заметающая прямая продолжает

опускаться, новая парабола расширяется. Часть новой параболы ниже старой линии приборя теперь становится частью новой линии приборя. Будем называть встречу заматающей прямой с новым центром *событием центра*.

Что происходит с диаграммой Вороного в момент события центра? Напомним, что точки излома на линии приборя вычерчивают ребра диаграммы Вороного. В момент события центра появляются две новые точки излома, которые начинают вычерчивать ребра. В начальный момент эти точки излома совпадают, а затем расходятся в разные стороны, вычерчивая одно и то же ребро. Сначала это ребро не связано с частью диаграммы Вороного над заматающей прямой. Но впоследствии — скоро мы увидим, когда именно, — растущее ребро наталкивается на другое ребро и соединяется с остальной частью диаграммы.

Итак, мы теперь понимаем, что происходит в момент события центра: на линии приборя появляется новая дуга и начинается вычерчивание нового ребра диаграммы Вороного. Может ли новая дуга появиться на линии приборя еще каким-то способом? Нет, не может.

Лемма 13.1. Единственная причина появления новой дуги на линии приборя — событие центра.

Доказательство. Предположим противное — что уже существующая парабола β_j , определенная центром p_j , «втискивается» в линию приборя. Это могло бы произойти двумя способами.

Первая возможность — β_j втискивается в середине дуги какой-то параболы β_i . В тот момент, когда это происходит, β_i и β_j касаются, т. е. имеют ровно одну общую точку. Обозначим l_y координату y заматающей прямой в момент касания. Если $p_j = (p_{j,x}, p_{j,y})$, то парабола β_j описывается уравнением

$$y = \frac{1}{2(p_{j,y} - l_y)}(x^2 - 2p_{j,x}x + p_{j,x}^2 + p_{j,y}^2 - l_y^2).$$

Уравнение для β_i , разумеется, аналогично. Пользуясь тем фактом, что $p_{j,y}$ и $p_{i,y}$ больше l_y , легко показать, что β_i и β_j не могут иметь только одну точку пересечения. Поэтому парабола β_j не может втиснуться в середину дуги другой параболы β_i .

Вторая возможность — β_j втискивается между двумя дугами. Пусть эти дуги являются частями парабол β_i и β_k . Обозначим q точку пересечения β_i и β_k , в которой β_j только-только появляется на линии приборя, и предположим, что β_i расположена на линии приборя левее q , β_k — правее q . Тогда существует окружность C , проходящая через p_i , p_j и p_k — центры, определяющие параболы. Эта окружность касается некоторой заматающей прямой l . Если начать с точки касания и двигаться по часовой стрелке, то точки будут расположены на окружности C в порядке p_i , p_j , p_k , поскольку мы

предположили, что β_j втискивается между дугами β_i и β_k . Рассмотрим бесконечно малое смещение заметающей прямой вниз при сохранении касания окружности C с l . Тогда не может быть так, что внутри C нет ни одного центра, и при этом она все-таки проходит через p_j . либо p_i , либо p_k окажутся внутри. Поэтому в достаточно малой окрестности q парабола β_j не может стать частью линии приборя при опускании заметающей прямой, т. к. либо p_i , либо p_k окажутся к l ближе, чем p_j .

Из этой леммы сразу следует, что количество параболических дуг в линии приборя не превышает $2n - 1$: каждый встретившийся центр порождает одну новую дугу и вызывает расщепление не более одной существующей дуги на две, а никак по-другому дуга появиться не может.

Второй тип событий в алгоритме заметания плоскости — схлопывание существующей дуги в точку и последующее исчезновение. Пусть α' — исчезающая дуга, а α и α'' — две соседних с α' дуги в момент, предшествующий ее исчезновению. Дуги α и α'' не могут быть частями одной и той же параболы; эту возможность можно исключить точно так же, как первую возможность в доказательстве последней леммы. Поэтому три дуги α , α' и α'' определены разными центрами p_i , p_j и p_k . В момент исчезновения α' параболы, определенные этими центрами, имеют общую точку q . Эта точка находится на равном расстоянии от прямой l и каждого из трех центров. Поэтому существует окружность, проходящая через p_i , p_j и p_k , с центром в точке q , самая нижняя точка которой лежит на l . Внутри этой окружности не может находиться ни одного центра диаграммы Вороного: такой центр был бы ближе к q , чем q к l , а это противоречит тому факту, что q расположена на линии приборя. Отсюда следует, что точка q — вершина диаграммы Вороного. Это не должно вызывать удивления, т. к. выше мы заметили, что точки излома на линии приборя вычерчивают ребра диаграммы Вороного. Таким образом, когда из линии приборя исчезает дуга и две точки излома сходятся, должны сойтись и ребра диаграммы. Будем называть событие, при котором заметающая прямая доходит до самой нижней точки окружности, проходящей через три центра, которые определяют соседние дуги на линии приборя, *событием окружности*. Из сказанного выше вытекает следующая лемма.

Лемма 13.2. Дуга может исчезнуть из линии приборя только в результате события окружности.

Итак, мы теперь знаем, когда и как изменяется комбинаторная структура линии приборя: в момент события центра появляется новая дуга, а в момент события окружности исчезает существующая. Мы также знаем, как это связано с конструируемой диаграммой Вороного: в момент события центра начинает расти новое ребро, а в момент события окружности два растущих

ребра встречаются и образуют вершину. Остается подобрать подходящие структуры данных для запоминания необходимой информации в процессе заметания. Наша цель — вычислить диаграмму Вороного, поэтому нужна структура, в которой будет храниться та часть, которая уже вычислена. Понадобятся также две «стандартные» структуры данных, применяемые в любом алгоритме заметания плоскости: очередь событий и структура, представляющая состояние заметающей прямой. В данном случае вторая структура будет содержать представление линии прибора. Ниже описана реализация всех структур данных.

- Конструируемую диаграмму Вороного мы будем хранить в обычной структуре данных, применяемой для разбиений, — двусвязном списке ребер. Но диаграмма Вороного — не настоящее разбиение; некоторые ее ребра являются полупрямыми или полными прямыми, а их представить в двусвязном списке ребер невозможно. В процессе построения это не составляет проблемы, потому что описанное ниже представление линии прибора позволит эффективно добираться до нужных частей двусвязного списка. Но по завершении построения нам хотелось бы иметь настоящий двусвязный список ребер. Для этого мы добавим большой прямоугольник, ограничивающий сцену — настолько большой, что содержит все вершины диаграммы Вороного. Тогда окончательное разбиение будет состоять из ограничивающего прямоугольника и находящейся внутри него части диаграммы Вороного.
- Линию прибора мы представим сбалансированным двоичным деревом поиска T ; это будет структура состояния. Его листья соответствуют дугам x -монотонной линии прибора в порядке следования: самый левый лист представляет самую левую дугу, следующий лист — вторую слева дугу и т. д. В каждом листе μ хранится центр, определяющий представленную этим листом дугу. Внутренние узлы T представляют точки излома линии прибора. Точка излома хранится во внутреннем узле в виде упорядоченного кортежа центров (p_i, p_j) где p_i определяет параболу слева от точки излома, а p_j — справа от нее. При таком представлении линии прибора мы можем найти дугу, расположенную выше нового центра, за время $O(\log n)$. Во внутреннем узле мы просто сравниваем координаты x нового центра и точки излома, причем последнюю можно вычислить за постоянное время, зная кортеж центров и позицию заметающей прямой. Отметим, что параболы в явном виде не хранятся.

В T мы храним также указатели на две другие структуры данных, используемые в процессе заметания. В каждом листовом узле T ,

представляющем дугу α , хранится один указатель на узел в очереди событий, а именно на тот, что представляет событие окружности, в момент которого α исчезает. Этот указатель равен NULL, если не существует такого события окружности, при котором α исчезает, или если это событие еще не наступило. Наконец, в каждом внутреннем узле v хранится указатель на полуребро в двусвязном списке ребер диаграммы Вороного. Точнее, в v хранится указатель на одно из полуребер того ребра, которое вычерчивается точкой излома, представленной v .

- Очередь событий Q реализована с помощью очереди с приоритетами, где приоритетом события является его координата y . В очереди хранятся грядущие события, о которых уже известно. В случае события центра мы храним просто сам центр. А для события окружности храним самую нижнюю точку окружности и указатель на листовой узел T , представляющий дугу, которая исчезнет в момент возникновения события.

Все события центров известны заранее, события окружностей — нет. И это подводит нас к последнему оставшемуся вопросу - обнаружению событий окружности.

В процессе заметания топологическая структура линии прибора изменяется при каждом событии. Иногда появляются новые тройки соседних дуг, а иногда исчезают существующие. Наш алгоритм должен гарантировать, что для любых трех соседних дуг на линии прибора, определяющих потенциальное событие окружности, это событие будет помещено в очередь Q . Тут есть две тонкости. Во-первых, могут существовать соседние тройки, для которых две точки излома не сходятся, т. е. они движутся в таких направлениях, которые исключают встречу в будущем. Это бывает, когда точки излома движутся вдоль срединных перпендикуляров, уводящих от точки пересечения. В таком случае тройка не определяет потенциальное событие окружности. Во-вторых, даже если точки излома тройки сходятся, соответствующее событие окружности может не произойти; это бывает, когда тройка исчезает (например, из-за появления нового центра на линии прибора) раньше, чем произойдет событие. Такое событие мы будем называть *ложной тревогой*.

Итак, вот что делает алгоритм. В момент каждого события он проверяет все вновь появившиеся тройки соседних дуг. Например, в момент события центра могут образоваться три новые тройки: в одной новая дуга находится слева, в другой - посередине, а в третьей — справа. Если у такой новой тройки точки излома сходятся, то событие помещается в очередь Q . Отметим, что в случае события центра тройка, в которой новая дуга находится посередине, никогда не приводит к событию окружности, потому что левая и правая

дуги тройки принадлежат одной и той же параболе, а потому точки излома должны расходиться. Далее, для каждой исчезающей тройки проверяется, находится ли соответствующее ей событие в очереди Q . Если да, то событие является ложной тревогой и удаляется из очереди. Это легко сделать с помощью хранящихся в листьях T указателей на соответствующие события в очереди Q .

Лемма 13.3. Каждая вершина диаграммы Вороного обнаруживается с помощью события окружности.

Доказательство. Для вершины q диаграммы Вороного пусть p_i, p_j и p_k — три центра, через которые проходит окружность $C(p_i, p_j, p_k)$, не содержащая внутри себя ни одного центра. Для простоты рассмотрим только случай, когда на окружности $C(p_i, p_j, p_k)$ нет других центров, а самая нижняя ее точка не совпадает ни с одной из точек p_i, p_j и p_k . Без потери общности можно предполагать, что, двигаясь от нижней точки $C(p_i, p_j, p_k)$ по часовой стрелке, мы встретим центры p_i, p_j и p_k именно в таком порядке.

Мы должны показать, что непосредственно перед тем, как заметающая прямая достигнет нижней точки $C(p_i, p_j, p_k)$, на линии прибора существуют три соседние дуги α, α' и α'' , определенные центрами p_i, p_j и p_k . Только в этом случае имеет место событие окружности. Рассмотрим положение заметающей прямой на бесконечно малом удалении от нижней точки $C(p_i, p_j, p_k)$. Поскольку ни на самой окружности $C(p_i, p_j, p_k)$, ни внутри нее нет других центров, то существует окружность, проходящая через p_i и p_j , которая касается заметающей прямой и не содержит внутри ни одного центра. Поэтому на линии прибора существуют соседние дуги, определяемые центрами p_i и p_j . Аналогично на линии прибора существуют соседние дуги, определяемые центрами p_j и p_k . Легко видеть, что обе дуги, определяемые центром p_j , на самом деле совпадают, а отсюда следует, что на линии прибора имеются три соседние дуги, определяемые центрами p_i, p_j и p_k . Поэтому соответствующее событие окружности находилось в Q непосредственно перед тем, как случиться, а, значит, вершина диаграммы Вороного обнаружена.

Теперь можно описать алгоритм заметания плоскости в деталях. Отметим, что после того как все события обработаны и очередь Q опустела, линия прибора еще не исчезла. Остались точки излома, соответствующие полубесконечным ребрам диаграммы Вороного. Как уже было сказано, в двусвязном списке ребер нельзя представить полубесконечные ребра, поэтому приходится добавить ограничивающий прямоугольник, к которому эти ребра можно будет присоединить. Ниже приведен псевдокод верхнего уровня алгоритма.

Вход. Множество $P := \{p_1, \dots, p_n\}$ точек на плоскости (центров).

Выход. Диаграмма Вороного $Vor(P)$ внутри ограничивающего прямоугольника, представленная в виде двусвязного списка ребер D .

1. Инициализировать очередь событий Q , содержащую все события центров, инициализировать пустую структуру состояния T и пустой двусвязный список ребер D .
2. **while** Q не пуста
3. **do** извлечь из Q событие с наибольшей координатой y
4. **if** это событие центра p
5. **then** $HandleSiteEvent(p)$
6. **else** $HandleCircleEvent(y)$, где y – листовый узел T , представляющий дугу, которая исчезнет
7. Внутренние узлы, оставшиеся в T , соответствуют полубесконечным ребрам диаграммы Вороного. Вычислить ограничивающий прямоугольник, содержащий внутри себя все вершины диаграммы Вороного, и присоединить полубесконечные ребра к этому прямоугольнику, соответствующим образом изменив двусвязный список ребер.
8. Обойти полуребра в двусвязном списке, чтобы добавить записи о ячейках и указатели на них и из них.

Процедуры обработки событий определены следующим образом.

$HandleSiteEvent(p_i)$

1. Если T пуста, то вставить в нее p_i (так что T будет состоять из единственного листового узла, в котором хранится p_i) и вернуться. В противном случае выполнить шаги 2–5.
2. Искать в T дугу α , находящуюся строго над p_i . Если в листовом узле, представляющем α , имеется указатель на событие окружности в Q , то это событие является ложной тревогой, и его следует удалить из Q .
3. Заменить листовый узел T , представляющий α , поддеревом с тремя листьями. В среднем листе хранится новый центр p_i , а в двух других — центр p_j , который раньше хранился вместе с α . Сохранить кортежи (p_i, p_j) и (p_j, p_i) , представляющие новые точки излома, в двух новых внутренних узлах. При необходимости выполнить перебалансировку T .
4. Создать в структуре, описывающей диаграмму Вороного, новые записи о полуребрах для ребра, разделяющего $V(p_i)$ и $V(p_j)$, которое будет вычерчиваться двумя новыми точками излома.
5. Проверить тройку соседних дуг, в которой новая дуга, соответствующая p_i , находится слева, и определить, сходятся ли точки излома. Если да,

то вставить в Q новое событие окружности и добавить указатели между узлом T и элементом Q . Сделать то же самое для тройки, в которой новая дуга находится справа.

HandleCircleEvent(γ)

1. Удалить из T листовой узел γ , который представляет исчезающую дугу α . Обновить кортежи, представляющие точки излома во внутренних узлах. При необходимости перебалансировать T . Удалить из Q все события окружности, имеющие отношение к α ; их можно найти с помощью указателей из узла T , предшествующего γ и следующего за ним. (Событие окружности, в котором α — средняя дуга, обрабатывается в данный момент и уже удалено из Q .)
2. Добавить центр окружности, породившей событие, в качестве записи о вершине в двусвязный список ребер D , в котором хранится конструируемая диаграмма Вороного. Создать две записи о полуребрах, соответствующие новой точке излома на линии приборя. Установить указатели между ними. Присоединить все три новые записи к записям о полуребрах, оканчивающихся в данной вершине.
3. Проверить новую тройку соседних дуг, в которой бывший левый сосед a является средней дугой, и определить, сходятся ли две точки излома этой тройки. Если да, то вставить в Q соответствующее событие окружности и установить указатели между новым событием в Q и соответствующим ему листовым узлом T . Сделать то же самое для тройки, в которой средней дугой является бывший правый сосед.

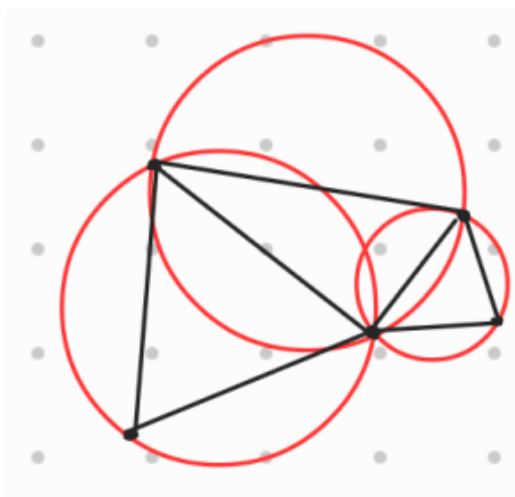
Лемма 13.4. Время работы описанного алгоритма составляет $O(n \log n)$, а размер потребляемой памяти равен $O(n)$.

Доказательство. Примитивные операции над деревом T и очередью событий Q , в частности вставка и удаление элемента, занимают время $O(\log n)$. Примитивные операции над двусвязным списком ребер занимают постоянное время. Для обработки события мы выполняем постоянное число таких примитивных операций, так что тратим на этом время $O(\log n)$. Очевидно, что всего существует n событий центра. Что касается событий окружности, заметим, что каждое такое обработанное событие определяет вершину $Vor(P)$. Напомним, что ложные тревоги удаляются из Q еще до обработки. Они создаются и удаляются в процессе обработки другого — настоящего — события, и затрачиваемое на них время учитывается во времени обработки этого события. Поэтому количество обрабатываемых событий окружности не превосходит $2n - 5$. Оценки времени и размера памяти доказаны.

14 Триангуляция Делоне, связь с диаграммами Вороного. Алгоритм построения

Триангуляция - максимальное планарное разбиение, к которому нельзя добавить ни одного ребра, соединяющего 2 вершины, без нарушения планарности.

Триангуляция Делоне - триангуляция для заданного множества точек S , при котором для каждого Δ -ка все точки из S , кроме вершин Δ -ка лежат вне окружности, описанной вокруг этого треугольника.



Триангуляция Делоне

Связь с диаграммой Вороного:

Теорема: (двойственность диаграммы Вороного и триангуляции Делоне)

Серединный перпендикуляр центров определяет ребро диаграммы Вороного \Leftrightarrow точка на этом перпендикуляре, такая что:

- граница круга (q) содержит только центры p_1 и p_2
- внутренность круга не содержит других центров мн-ва P

Получение диаграммы Вороного из триангуляции Делоне

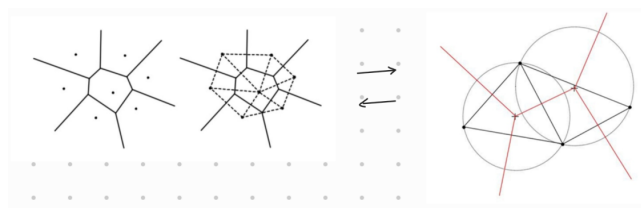


Диаграмма Вороного — триангуляция Делоне

Алгоритм построения триангуляции Делоне:

Наивный

1. Перебираем все треугольники (все тройки точек из S) ($O(n^3)$)
2. Описываем вокруг каждого треугольника окружность $O(1)$ и проверяем, пуста ли каждая окружность. Те Δ -ки, окружности которых являются пустыми, объявляются гранями триангуляции Делоне (T) - для одной $\Rightarrow (T_i)$
3. Перебираем все пары выбранных Δ -ов и проверяем на смежность $O(n^2)$

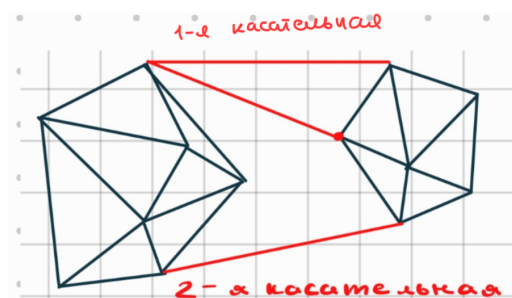
Итого: $O(n^4)$

Триангуляция методом "разделяй и властвуй":

- делим множество точек на мелкие множества (рекурсивно)
- затем объединять оптимальные триангуляции

"удаляй и строй": объединение 2-х триангуляций

1. Ищем 2 общие касательные (верхняя и нижняя). Верхнюю обозначаем за базовую линию.
2. Ищем внешний узел, ближайший к базовой линии и строим треугольник по базовой линии и этому узлу.
3. Берём новую сторону Δ -ка за базовую линию, повторяем процесс до нижней касательной.

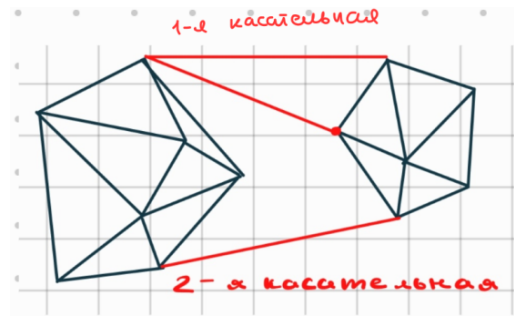


или

"строй и перестраивай"

1. Ищем 2 общие касательные (верхняя и нижняя). Верхнюю обозначаем за базовую линию.
2. Рассматриваем 2 ближайшие к базовой прямой точки строим 2 Δ -ка

3. Выбираем тот треугольник, \min угол которого больше (\Rightarrow этот треугольник более равнобедренный и будет перестроен с меньшей вероятностью)
4. Берём новую сторону Δ -ка за базовую линию, повторяем процесс до 2-й касательной.
5. Проверка всех Δ -ов на условие Делоне.



Сложность алгоритма:

- разбиение множества точек $O(\log n)$
- каждое объединение $O(n)$

Всего: $O(n \log n)$

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

15.1 Граф видимости

Рассмотрим точное решение нахождения кратчайшего пути на плоскости между двумя точками с полигональными препятствиями с помощью построения графа видимости. После его построения кратчайший путь ищется любым стандартным алгоритмом поиска (например, алгоритмом Дейкстры или A^*).

Для простоты рассуждений начальную и конечную вершины будем считать вершинами полигонов.

Лемма 15.1. Любой кратчайший путь между двумя вершинами с полигональными препятствиями представляет собой ломаную, вершины которой — вершины полигонов.

Доказательство. Пусть кратчайший путь — не ломаная. В таком случае, на пути существует такая точка p , которая не принадлежит ни одному прямому отрезку. Это означает, что существует ε -окрестность точки p , в которую не попадает ни одно препятствие (случай, когда точка попала на ребро рассматривается аналогично). В таком случае, подпуть, который находится внутри ε -окрестности, по неравенству треугольника может быть сокращён по хорде, соединяющей точки пересечения границы ε -окрестности с путем. Раз часть пути может быть уменьшена, значит и весь путь может быть уменьшен, а значит исходное предположение некорректно.

Опр. 15.1. Говорят, что вершина u видна (англ. mutually visible) из v , если отрезок uv не пересекает ни одного препятствия.

Опр. 15.2. Граф видимости (англ. visibility graph) — граф, вершины которого — вершины полигонов. Между вершинами u и v существует ребро, если из u видна v .

В худшем случае в таком графе может быть $O(n^2)$ ребер. Однако по некоторым ребрам кратчайший путь точно не пройдет, и такие ребра из графа можно удалить.

Лемма 15.2. 1. Если существуют вершины A, B, C одного препятствия и вершина D такая, что поворот DBA не совпадает с поворотом DBC , то ребро DB не принадлежит кратчайшему пути и его можно удалить из графа.

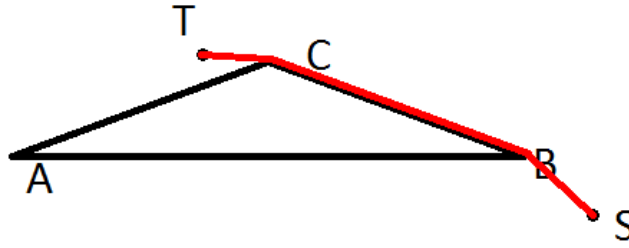


Figure 21: Не удаляем BS

2. Все внутренние вершины, кроме вырожденного случая, (начальная/конечная точка лежит внутри выпуклой оболочки фигуры) можно игнорировать.

Доказательство.

1. Путь проходящий через ребро BD будет длиннее, чем через соседей точки B , так как по неравенству треугольника $AB + BD > AD$.
2. Если случай не вырожденный, значит заход внутрь фигуры только увеличит суммарный путь, так как по неравенству треугольника расстояние между соседними выпуклыми вершинами всегда меньше суммы расстояний с учётом внутренней.

По доказанным леммам любое ребро кратчайшего пути содержится в графе. Таким образом, для нахождения кратчайшего пути осталось найти кратчайший путь в этом графе от начальной до конечной вершины.

15.2 Планирование движения

Рассмотрим задачу нахождения кратчайшего пути, когда движимый объект — это выпуклый полигон. Например, робот, которого надо доставить из начальной в конечную точку.

Если полигон вращать нельзя, задачу сводится к движению точки так: выбирается точка на полигоне, которая принимается за начало координат. В такой системе координат для каждого препятствия считается сумма Минковского с полигоном. Получаются бОльшие препятствия, но теперь достаточно двигать выбранную точку, что было описано выше.

15.3 Сумма Минковского

Каждой точке (x, y) ставится в соответствие фигура робота $R(x, y)$ с точкой привязки, помещенной в точку (x, y) .

Опр. 15.3. Для заданного робота R и препятствия P , -препятствием называется множество точек, будучи помещенным в которые, робот заденет препятствие: $CP := \{(x, y) : R(x, y) \cap P \neq \emptyset\}$.

Опр. 15.4. Суммой Минковского двух множеств $S_1 \subset R^2$, $S_2 \subset R^2$ называется множество $S_1 \oplus S_2 := \{p + q : p \in S_1, q \in S_2\}$, где $p + q$ обозначает векторную сумму.

Опр. 15.5. Отрицанием множества $S \subset R^2$ называется множество $-S := \{-p : p \in S\}$, где $-p$ обозначает векторное отрицание.

Теорема 15.1. Для заданного робота R и препятствия P , -препятствием является множество $P \oplus (-R(0, 0))$.

Доказательство. Необходимо доказать, что робот $R(x, y)$ пересекает препятствие P в том и только в том случае, если $(x, y) \in P \oplus (-R(0, 0))$.

Пусть робот задевает препятствие, и точка $q = (q_x, q_y)$ является точкой пересечения. Тогда, так как $q \in R(x, y)$, то $(q_x - x, q_y - y) \in R(0, 0)$, или $(x - q_x, y - q_y) \in -R(0, 0)$. А заметив, что $q \in P$, получаем $(x, y) \in P \oplus (-R(0, 0))$.

В обратную сторону, пусть $(x, y) \in P \oplus (-R(0, 0))$, тогда существуют точки $(r_x, r_y) \in R(0, 0)$ и $(p_x, p_y) \in P$ такие, что $(x, y) = (p_x - r_x, p_y - r_y)$ или $(p_x, p_y) = (x + r_x, y + r_y)$, а это означает, что $R(x, y)$ пересекает P .

Теорема 15.2. Пусть заданы две выпуклые фигуры P и R с числом вершин n и m соответственно. Тогда суммой Минковского $P \oplus R$ является выпуклая фигура с не более чем $m + n$ вершинами.

Доказательство. Для начала заметим, что любая крайняя точка в направлении вектора \vec{d} есть сумма крайних точек фигур в этом направлении. Убедиться в этом можно, спроецировав обе фигуры на вектор \vec{d} .

Теперь рассмотрим произвольное ребро e из $P \oplus R$. Оно является крайним в направлении к своей нормали, а значит оно образовано крайними точками фигур, и хотя бы у одной из фигур должно быть ребро, которое является крайним в этом направлении. Сопоставим e с этим ребром. Тогда, сопоставив таким образом всем ребрам $P \oplus R$ ребра исходных фигур, получаем, что всего ребер в $P \oplus R$ не более чем $m + n$, так как каждое ребро исходных фигур использовалось не более раза.

15.3.1 Псевдокод

```
i = j = 0
V[n] = V[0], V[n+1] = V[1], W[m] = W[0], W[m+1] = W[1]
while i < n or j < m do
    add V[i]+W[j] to answer
    if angle(V[i], V[i+1]) < angle(W[j], W[j+1])
        ++i
    else if angle(V[i], V[i+1]) > angle(W[j], W[j+1])
        ++j
```

```
else
    ++i, ++j
```

15.3.2 Случай невыпуклых фигур

Для начала заметим следующий факт: $S_1 \oplus (S_2 \cup S_3) = (S_1 \oplus S_2) \cup (S_1 \oplus S_3)$.

В случае, когда одна из фигур невыпукла, её сначала надо затриангулировать, получив $n - 2$ треугольников. После этого, уже известным алгоритмом, надо построить $n - 2$ выпуклых фигур с не более чем $m + 3$ вершинами, которые будут суммами Минковского соответствующих треугольников. Объединение этих выпуклых фигур будет состоять из $O(nm)$ вершин.

В случае, когда обе фигуры невыпуклы, обе эти фигуры надо затриангулировать, получив $n - 2$ и $m - 2$ треугольников соответственно. Построив суммы Минковского множеств этих треугольников, получим $(n - 2)(m - 2)$ выпуклых фигур, объединение которых состоит из $O(n^2m^2)$ вершин.

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

Задача:

Дана сцена с 2D или 3D объектами и наблюдатель, который смотрит на сцену из своей точки обзора. Нужно отрисовать на сцене видимые наблюдателю части объектов.

16.1 Алгоритм z-буфера (z-buffer algorithm)

Для **отсечения невидимых частей поверхности** существует простой, но длительный метод — алгоритм z-буфера. В направлении просмотра проводится ось z-координат, затем определяется, какие пиксели покрывают проекции объектов. Алгоритм хранит информацию об уже обработанных объектах в двух буферах: буфере кадра и z-буфере.

- В буфере кадра для каждого пикселя хранится информация о цвете объекта, отображаемого им на данный момент.
- В z-буфере для каждого пикселя хранится z-координата видимого на данный момент объекта, точнее, в нем хранится z-координату точки такого объекта.

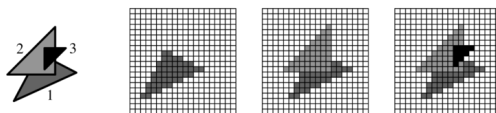
Предположим, что мы выбрали пиксель и преобразовываем объект.

- Если z-координата объекта в этом пикселе меньше, чем z-координата, хранимая в z-буфере, тогда новый объект лежит перед видимым на данный момент (меняем z-буфер и буфер кадра)
- Если z-координата объекта в этом пикселе больше, чем z-координата, хранимая в z-буфере, то новый объект не видим, и буферы останутся без изменений.

Алгоритм z-буфера легко реализовать, и он быстро работает, но требует много дополнительной памяти

Алгоритм художника (painter's algorithm)

Алгоритм художника избегает дополнительных затрат памяти, изначально сортируя объекты по расстоянию от них до точки обзора. Тогда объекты проверяются в **порядке глубины**, начиная от самого дальнего.



Пример работы алгоритма художника

Основная проблема: порядок глубины не всегда существует: отношение "перед" может содержать циклы. Когда такое циклическое перекрытие происходит, объекты не могут быть корректно отсортированы.

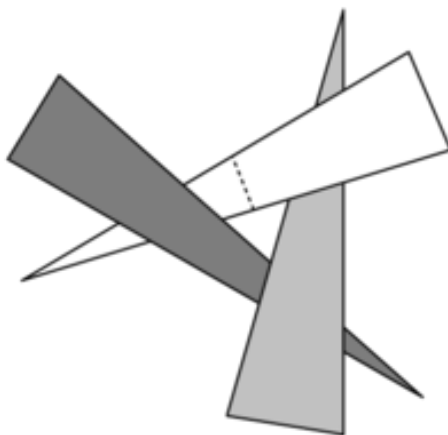


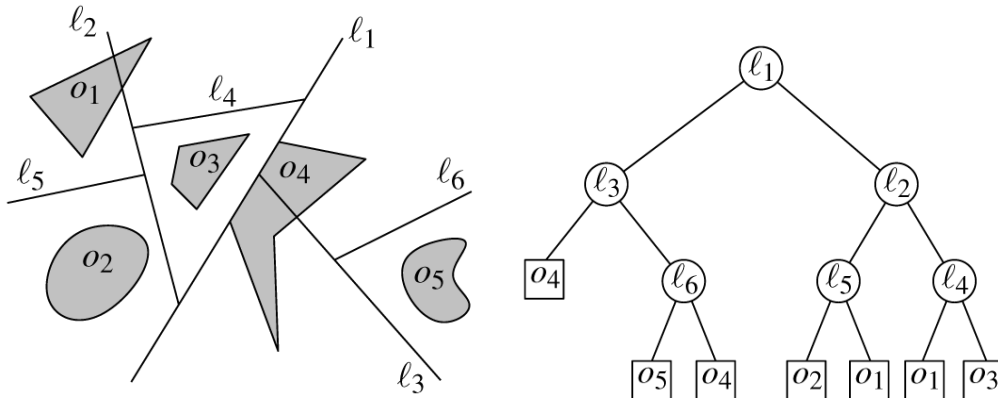
Иллюстрация циклического наложения фигур — проблемы данного алгоритма

Решение: сортировка фрагментов объектов (сначала — разбиение объектов на фрагменты)

Задачу разбиения на фрагменты можно решить с помощью **двоичного разбиения пространства** (англ. binary space partitioning, BSP)

16.2 BSP-деревья

Покажем двоичное разбиение пространства на примере рисунка ниже



Прямые разбивают на части не только плоскость, но и объекты, расположенные на ней. Разбиение продолжается до тех пор, пока внутри каждой грани плоскости окажется не более одного фрагмента объекта.

Этот процесс можно представить с помощью двоичного дерева. Каждый лист дерева соответствует **грани разбиения**, в нем хранится фрагмент объекта, находящийся внутри этой грани. Каждый узел дерева соответствует **разбивающей прямой**, которая хранится в этом узле.

Опр. 16.1. BSP-дерево (англ. binary space partition tree) — дерево, отвечающее заданному двоичному разбиению пространства.

Свойства BSP-деревьев

Пусть гиперплоскость h делит пространство на h^+ и h^- . Пусть S — множество объектов, для которого мы строим разбиение в d -мерном пространстве, v — какая-то вершина дерева, тогда обозначим за $S(v)$ множество объектов (возможно пустое), хранимых в этой вершине.

- Если $|S| \leq 1$, то T — лист. Фрагмент объекта в S , если он существует, хранится в этом листе.
- Если $|S| > 1$, то в корне дерева v хранится гиперплоскость h и множество $S(v)$ объектов, которые полностью содержатся в h .

BSP-деревья и алгоритм художника

Предположим, что мы построили BSP-дерево T для множества объектов S в трехмерном пространстве.

Пусть p_{view} — точка обзора, и она лежит над разбивающей плоскостью, хранимой в корне T .

Тогда ни один из объектов, лежащих под этой плоскостью, не может перекрыть ни один из объектов, лежащих выше нее. Таким образом, мы можем безопасно отрисовать фрагменты объектов из поддерева T^- до отрисовки объектов из поддерева T^+ . Порядок фрагментов объектов в поддеревьях определяется таким же способом.