1. Алгоритм. Память и время как ресурсы.

Алгоритм

Алгоритм - это набор шагов или инструкций, необходимых для решения задачи
ИЛИ
Алгоритм - это набор конечного числа правил, задающих последовательность выполнения операций компьютерной программой для решения задачи определённого типа

Свойства алгоритма:

- **Конечность.** Алгоритм всегда должен заканчиваться после выполнения конечного числа шагов.
- **Определённость.** Действия, которые нужно выполнить, должны быть строго и недвусмысленно определены для каждого возможного случая.
- **Вввод.** Алгоритм имеет некоторое (возможно, равное нулю) число входных данных.
- **Вывод.** У алгоритма есть одно или несколько выходных данных , т. е. величин, имеющих вполне определенную связь с входными данными.
- **Эффективность.** Алгоритм обычно считается эффективным, если все его операторы достаточно просты для того, чтобы их можно было точно выполнить в течение конечного промежутка времени с помощью карандаша и бумаги.

Память

Для анализа алгоритма обычно используется анализ пространственной сложности алгоритма, чтобы оценить необходимую память времени исполнения как функцию от размера входа. Результат обычно выражается в терминах **«О» большое**.

Существует **четыре аспекта использования памяти**:

- Количество памяти, необходимой для **хранения кода алгоритма.**
- Количество памяти, необходимое для **входных данных**.
- Количество памяти, необходимое **для любых выходных данных** (некоторые алгоритмы, такие как сортировки, часто переставляют входные данные и не требуют дополнительной памяти для выходных данных).
- Количество памяти, необходимое **для вычислительного процесса** во время вычислений (сюда входят именованные переменные и любое стековое пространство, необходимое для вызова подпрограмм, которое может быть существенным при использовании рекурсии).

Время

Для анализа алгоритма обычно используется анализ временной сложности алгоритма, чтобы оценить время работы как функцию от размера входных данных. Результат обычно выражается в терминах «О» большое

> Не выражается в секундах, минутах или часах, так как эти величины зависят от мощности железа, на котором запускается алгоритм. Поэтому время считают в количестве **итераций (операций)**.

---

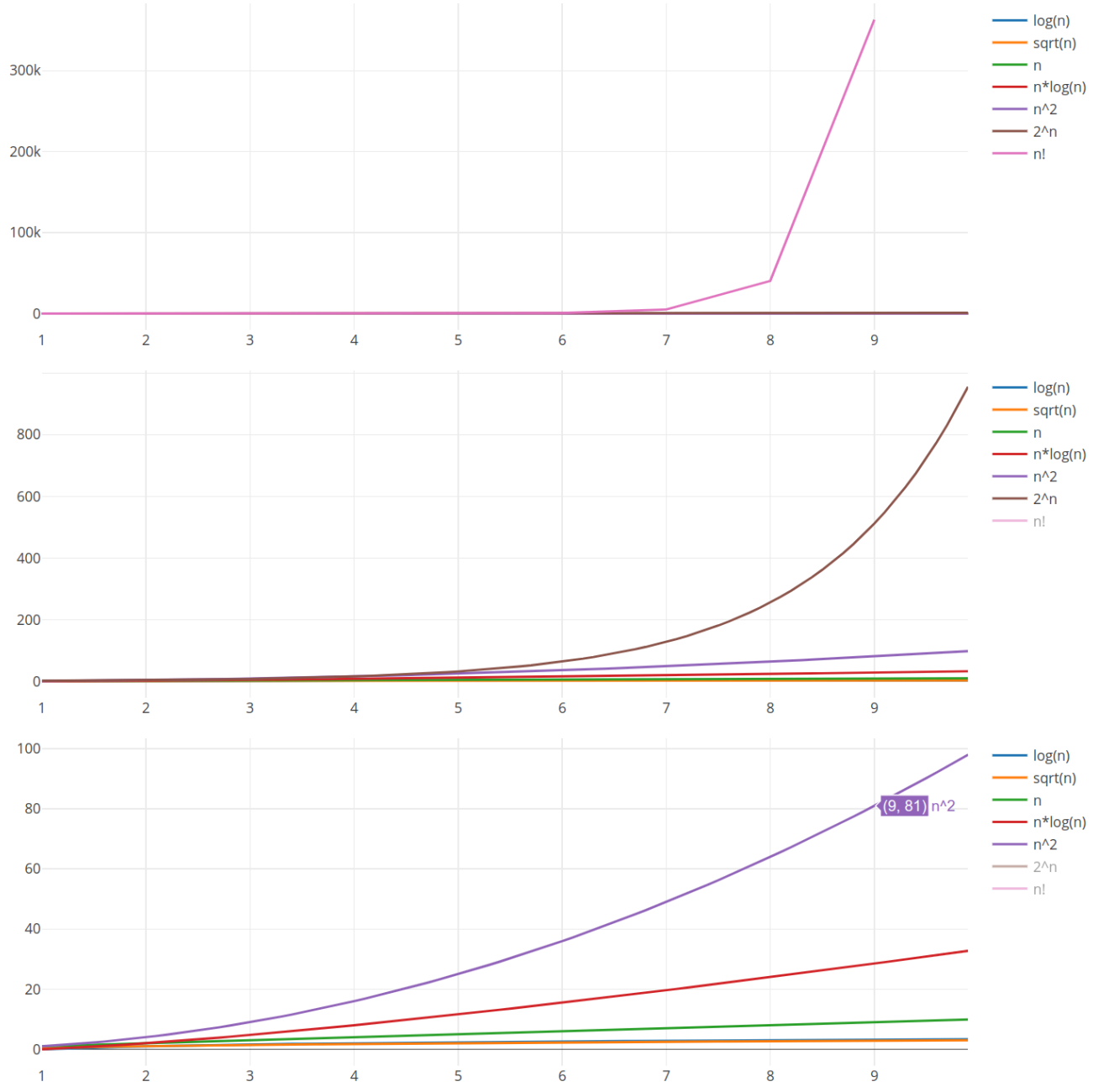
2. О-символика как инструмент оценки ресурсов, различные асимптотики (логарифм, полином, экспонента).

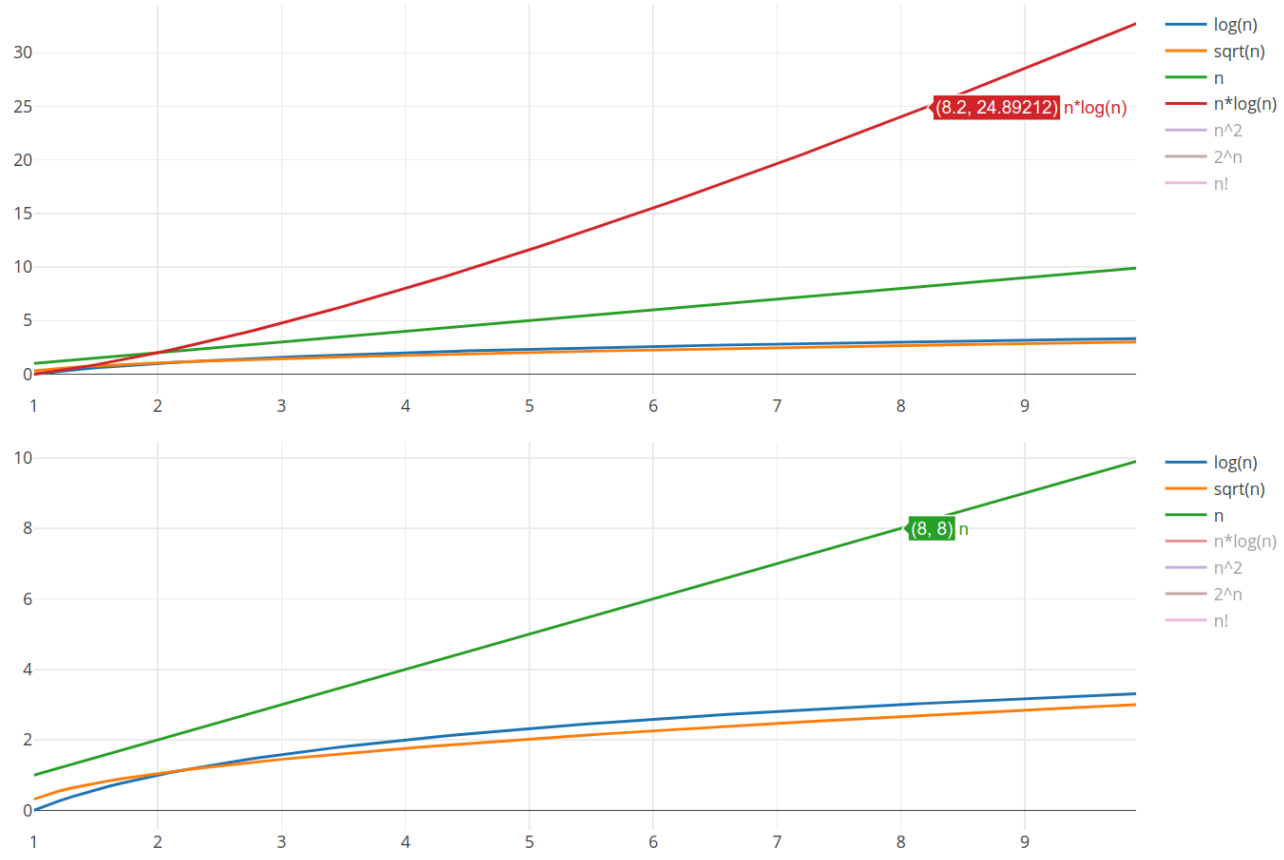
"О" большое - Это математическое обозначение для сравнения асимптотического поведения (асимптотики) функций. Используются в различных разделах математики, но активнее всего — в математическом анализе, теории чисел и комбинаторике, а также в информатике и теории алгоритмов. Под асимптотикой понимается характер изменения функции при стремлении её аргумента к определённой точке.

Запись $O(f(n))$ означает, что с увеличением параметра, характеризующего количество входной информации алгоритма, время работы алгоритма будет возрастать не быстрее, чем $f(n)$, умноженная на некоторую константу.

$f(n) = O(1)$ константа
$f(n) = O(log(n))$ логарифмический рост
$f(n) = O(n)$ линейный рост
$f(n) = O(n * log(n))$ квазилинейный рост
$f(n) = O(n^m)$ полиномиальный рост
$f(n) = O(2^n)$ экспоненциальный рост - самый худший
$f(n) = O(n!)$ сначала нормально, потом резко растет

$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n \prec n!$$





---

3. Метод математической индукции, использование для доказательства оценок

Математическая индукция - если утверждение истинно в одном случае, то оно окажется истинным и в следующем за ним случае.

Математическая индукция — это метод доказательства, который состоит из двух основных шагов:

1. **База индукции**: Необходимо показать, что утверждение $P(1)$ верно.
2. **Шаг индукции**: Нужно доказать, что если утверждение верно для некоторых целых положительных чисел $P(1), P(2), \ldots, P(n)$, то оно также верно для следующего числа $P(n+1)$. Это доказательство должно быть справедливо для любого целого положительного n.

> P.S. *Не очень понятно, что конкретно нужно*

---

4. Алгоритмы для работы с большими числами: сложение, умножение, быстрое

возведение в степень

Необходимость использования:

- Ограничения на объем типов данных
- Время выполнения операций на больших числах
- Использование встроенных функций ограниченно типами данных и особенностями реализации

Сложение

1. Числа представлены в виде массивов, состоящих из "коротких" чисел (играют роль цифр)
2. Массив для результата и переменная для переноса (изначально 0)
3. Складываются разряды двух чисел с учётом переноса
4. Если после сложения всех разрядов остался перенос, он добавляется вперёд числа (к старшему разряду)

Умножение большого числа на малое

Аналогично сложению, только вместо сложения разрядов(шаг 3) происходит перемножение, и также учитывается сложение с переносом.

Умножение большого числа на большое

Состоит из умножения на малое число и сложения.

1. Одно число умножается на каждый разряд другого (умножение большого числа на малое)
2. Промежуточные результаты складывается (учитывая "положения числа", т. е. 25 * 25 это 125 + 50***0***)

Быстрое возведение в степень

1. Если степень n чётная, основание возводится в квадрат, а степень делится на 2.
2. Если степень n нечётная, основание умножается на результат, а степень уменьшается на 1.
3. Процесс повторяется до тех пор, пока k>0.

---

5. Арифметика по модулю: сложение, умножение, возведение в степень

Если два целых числа a и b при делении на m дают одинаковые остатки, то они называются **сравнимыми (или равноостаточными) по модулю числа m. a ≡ b (mod m)**
сравнимыми (или равноостаточными) по модулю числа m. a ≡ b (mod m)

Операции по модулю:
(A + B) mod C = (A mod C + B mod C) mod C
(A *B) mod C = (A mod C* B mod C) mod C
A^B mod C = ((A mod C)^B) mod C

---

6. Алгоритм Евклида, расширенный алгоритм Евклида.

Алгоритм Евклида - находит **наибольший общий делитель** для чисел m и n

1. Разделим m на n, и пусть остаток от деления будет равен r (где $0 \leq r < n$).
2. Если $r = 0$, то выполнение алгоритма прекращается; n — искомое значение.
3. Присвоить $m = n, n = r$ и вернуться к шагу 1.

```
def euclidus(m, n):
    while n:
        m, n = n, m % n
    return m
```

Расширенный алгоритм Евклида
В то время как "обычный" алгоритм Евклида просто находит наибольший общий делитель двух чисел m и n, расширенный алгоритм Евклида находит помимо НОД также коэффициенты x и y такие, что:

$$x * m + y * n = НОД(m, n)$$

Т.е. он находит коэффициенты, с помощью которых НОД двух чисел выражается через сами эти числа.

```
def extended_euclidus(m, n):
    if n == 0:
        return m, 1, 0
    d, x1, y1 = extended_euclidus(n, m % n)
    x = y1
    y = x1 - (m // n) * y1
    return d, x, y
```

---

7. Проверка чисел на простоту, решето Эратосфена.

Простое число - число, которое делится нацело только на себя и на 1.

Соответственно, чтобы проверить число n на простоту достаточно перебрать все числа от 2 до числа n. Можно ускорить процесс, перебрав числа только до *корня n*.

> Число 64 делится на 2, 4, 8, 16, 32.
> 64 = 8. Число 8 делится на 2, 4.
> Всё дело в том, что если расписывать число 64 как произведение двух множителей, то получим следующую ситуацию:

```
2 * 32 = 64
4 * 16 = 64
8 * 8 = 64
16 * 4 = 64
32 * 2 = 64
```

Нетрудно видеть, что после 8 * 8 = 64 варианты повторяются. А 8 - как раз корень из числа 64. Поэтому достаточно проверить лишь числа до n включительно, так как корень из числа, если окажется целым, точно будет делителем числа (очевидно почему).

```
def is_prime(n):
    for i in range(2, int(n**0.5)+1):
        if not n % i:
            return False
    return True
```

Решето Эратосфена - алгоритм нахождения всех простых чисел до некоторого целого числа n.
Принцип простой: строим список чисел от 2 до n, после чего удаляем из него все составные числа.
Составное число - число, имеющее делители помимо 1 и числа n. Т.е. все числа, кроме простых.
Если более подробно, то алгоритм построения решета Эратосфена выглядит так:
Выписать подряд все целые числа от двух до n (2, 3, 4, …, n).
Пусть переменная p изначально равна двум — первому простому числу.
Зачеркнуть в списке числа от 2p до n, считая шагами по p (это будут числа, кратные p: 2p, 3p, 4p, …).
Найти первое незачёркнутое число в списке, большее чем p, и присвоить значению

переменной p это число.
Повторять шаги 3 и 4, пока возможно.

```
def eratosthenes(n):
    arr = [i for i in range(n + 1)]
    arr[1] = 0
    for p in arr:
        if p != 0:
            arr[2*p::p] = [0] * len(arr[2*p::p])
    return [i for i in arr if i != 0]
```

---

8. Криптография: схемы с закрытым ключом, RSA.

RSA (аббревиатура от фамилий Rivest, Shamir и Adleman) — криптографический алгоритм с открытым ключом, основывающийся на вычислительной сложности задачи факторизации больших полупростых чисел.

Например, 592939 * 592967 = 351593260013. Но как имея только число 351593260013 узнать числа 592939 и 592967? Это называется «сложность задачи факторизации произведения двух больших простых чисел», т.е. в одну сторону просто, а в обратную невероятно сложно.

RSA использует два ключа — открытый (публичный) и закрытый (приватный). Открытый ключ может быть доступен всем и используется для шифрования сообщений. Закрытый ключ хранится в секрете и используется для расшифровки полученных сообщений

Генерация ключей

1. Выбираем два случайных простых числа p и q
2. Вычисляем их произведение: N = p * q
3. Вычисляем функцию Эйлера: φ(N) = (p-1) * (q-1)
4. Выбираем число e (обычно простое, но необязательно), которое меньше φ(N) и является взаимно простым с φ(N) (не имеющих общих делителей друг с другом, кроме 1).
5. Ищем число d, обратное числу e по модулю φ(N). Т.е. остаток от деления $(d * e)$ и φ(N) должен быть равен 1. Найти его можно через расширенный алгоритм Евклида.
   e и n – открытый ключ
   d и n – закрытый ключ

9. Квадратичные сортировки (вставками, выбором минимума).

Пузырьковая

Алгоритм состоит из повторяющихся проходов по сортируемому массиву.

За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется перестановка элементов.

Проходы по массиву повторяются $N - 1$ раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован.

> *Сложность:* $O(n^2)$

```python
def bubble_sort(arr: list) -> list:
    for j in range(len(arr) - 1):
        f = False
        for i in range(len(arr) - 1 - j):
            if arr[i] > arr[i+1]:
                arr[i], arr[i+1] = arr[i+1], arr[i]
                f = True
        if not f:
            break
```

Вставками

В начальный момент отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан.

Данный алгоритм можно ускорить при помощи использования бинарного поиска для нахождения места текущему элементу в отсортированной части. Проблема с долгим сдвигом массива вправо решается при помощи смены указателей.

> *Сложность:* $O(n^2)$

```python
def insert_sort(arr):
    for i in range(1, len(arr)):
        j = i
        while j > 0 and arr[j] < arr[j-1]:
            arr[j], arr[j-1] = arr[j-1], arr[j]
            j -= 1
```

С бинарным поиском

```
def insertion_sort(arr: list) -> list:
    for i in range(1, len(arr)):
        left, right = 0, i - 1
        mid = (left + right) // 2

        while left < right:
            if arr[i] == arr[mid]:
                break
            elif arr[i] < arr[mid]:
                right = mid
            else:
                left = mid + 1
            mid = (left + right) // 2

        if arr[i] > arr[mid]:
            mid += 1

        arr[mid], arr[mid + 1: i + 1] = arr[i], arr[mid:i]
```

---

10. Метод «разделяй и властвуй». Бинарный поиск.

Метод "разделяй и властвуй"

«**Разделяй и властвуй»** в информатике — схема разработки алгоритмов, заключающаяся в рекурсивном разбиении решаемой задачи на две или более подзадачи того же типа, но меньшего размера, и комбинировании их решений для получения ответа к исходной задаче; разбиения выполняются до тех пор, пока все подзадачи не окажутся элементарными. Применяется в таких алгоритмах как бинарный (двоичный) поиск и сортировка слиянием.

Бинарный поиск

Дана таблица записей $R_1, R_2, \ldots, R_N$, ключи которых расположены в порядке возрастания: $K_1 < K_2 < \ldots < K_N$; алгоритм используется для поиска в таблице заданного аргумента K.

1. Установить $l \leftarrow 1, u \leftarrow N$.
2. Если $u < l$, алгоритм завершается неудачно; иначе установить $i \leftarrow floor((l+u)/2)$, чтобы i соответствовало примерно середине рассматриваемой части таблицы.
3. Если $K < K_i$, перейти к шагу 4; если $K > K_i$, перейти к шагу 5, если $K = K_i$, алгоритм успешно завершается.
4. Установить $u \leftarrow i - 1$ и перейти к шагу 2.
5. Установить $l \leftarrow i + 1$ и перейти к шагу 2.

---

11. Сортировка слиянием: наивная и эффективная реализация.

Суть:

1. Сортируемый массив разбивается на две части примерно одинакового размера. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).
2. Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
3. Два упорядоченных массива половинного размера соединяются в один.
    1. На каждом шаге мы берем меньший из двух первых элементов подмассивов и записываем его в результирующий массив. Счётчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваем на 1.
    2. Когда один из подмассивов закончился, мы добавляем все оставшиеся элементы второго подмассива в результирующий массив.

Подвид наивной реализации, сверху-вниз:

```
def top_down_merge_sort(A):
    if len(A) == 1:
        return A

    d = len(A) // 2
    left = top_down_merge_sort(A[:d])
    right = top_down_merge_sort(A[d:])

    return merge(left, right)
```

Еще подвид наивной реализации, снизу-вверх:

```
def bottom_up_merge_sort(A):
    k = 1
    while k < len(A):
        for i in range(0, len(A)-k, 2*k):
            A[i:i+2*k] = merge(A[i:i+k], A[i+k:i+2*k])
        k *= 2

    return A
```

Наивная функция слияния

```
 def merge(A, B):
    i, j, C = 0, 0, []
    while True:
        if A[i] < B[j]:
            C.append(A[i])
            i += 1
            if i == len(A):
                C.extend(B[j:])
                break
        else:
            C.append(B[j])
            j += 1
            if j == len(B):
                C.extend(A[i:])
                break
    return C
```

Галопирование — это оптимизация, используемая в алгоритме сортировки слиянием, которая позволяет ускорить процесс слияния двух отсортированных массивов.
Использует видоизменённый бинарный поиск:
Пусть нам даны отсортированный массив $B = (b_i : i < m)$ и элемент x. Надо найти $i < m$ такое, что $bi \leqslant x \leqslant bi + 1$, если $i + 1 < m$.

1. $k = 0$
2. пока $2k - 1 < m$ и $2k - 1 < x$ увеличиваем k на 1;
3. если $2k - 1 \geqslant m$, то $l = m - 1$, иначе $l = 2k - 1$;
4. пока $l \geqslant 0$ и $l > x$ уменьшаем l на 1;
5. $l + 1$ — искомая позиция;

```
def galloping(AB, n, C):
    C[:] = AB[:n]
    # копируем в массив C массив A

    # AB: объединенный массив, содержащий два
```

```
    # отсортированных подмассива (первый из A, второй из B).

    # n: длина первого подмассива A.
    # C - ?

    # r -- указатель на конец результата
    # j -- место последней вставки
    # m -- длина остатка B
    r, j, m = 0, n, len(AB) - n
    for i in range(n):
        # k -- степень двойки
        # l -- указатель на 2^k-1 элемент
        k, l = 0, 0
        while l < m and AB[j+l] < C[i]:
            k += 1
            l = 2**k - 1

        if l >= m:
            l = m - 1

        while l >= 0 and AB[j+l] > C[i]:
            l -= 1

        l += 1
        AB[r:r+l], AB[r+l] = AB[j:j+l], C[i]
        r, j, m = r + l + 1, j + l, m - l
```

Chunking — это метод, используемый для разделения последовательности данных на более мелкие, управляемые части или "чанки". В контексте представленного кода функция chunking разбивает отсортированный массив на участки, где каждый участок представляет собой последовательность элементов, расположенных в порядке возрастания (убывания).

```
def chunking(A):
    chunks = []
    a, d = 0, 0
    for b in range(1, len(A)):
        if d == 0:
            d = A[b] - A[a]
            continue

        if (A[b] - A[b-1])*d < 0:
            chunks.append((a, b-1) if d > 0 else (b-1, a))
            a, d = b, 0

    chunks.append((a, b) if d > 0 else (b, a))

    return chunks
```

12. Быстрая сортировка: понятие вероятностного алгоритма, время работы в среднем, простейший алгоритм, inplace-алгоритм.

Быстрая сортировка, сортировка Хоара (quicksort, qsort (по имени в стандартной библиотеке языка Си) — алгоритм сортировки, разработанный английским информатиком Тони Хоаром во время своей работы в МГУ в 1960 году.

Алгоритм:

1. Массив $a[l \dots r]$ разбивается на два (возможно пустых) подмассива $a[l \dots q]$ и $a[q+1 \dots r]$ таких, что каждый элемент $a[l \dots q]$ меньше или равен $a[q]$, который в свою очередь, не превышает любой элемент подмассива $a[q+1 \dots r]$. Индекс вычисляется в ходе процедуры разбиения.
2. Подмассивы $a[l \dots q]$ и $a[q+1 \dots r]$ сортируются с помощью рекурсивного вызова процедуры быстрой сортировки.
3. Поскольку подмассивы сортируются на месте, для их объединения не требуются никакие действия: весь массив $a[l \dots r]$ оказывается отсортированным.

Вероятностный алгоритм – это такой алгоритм, который использует случайность для принятия решений. В случае быстрой сортировки это означает, что выбор опорного элемента (pivot) происходит почти случайно. Вероятностная природа позволяет избежать худших случаев при работе алгоритма.

Простейшая реализация

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[0]
    left = []
    right = []

    for x in arr[1:]:
        if x < pivot:
            left.append(x)
        else:
            right.append(x)

    return quicksort(left) + [pivot] + quicksort(right)
```

In-place

```
def partition(array, low, high):
    pivot = array[high]
    i = low - 1

    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            array[i], array[j] = array[j], array[i]

    array[i + 1], array[high] = array[high], array[i + 1]
    return i + 1


def quicksort(array, low, high):
    if low < high:
        pi = partition(array, low, high)

        quicksort(array, low, pi - 1)
        quicksort(array, pi + 1, high)
```

Сложность:

- в среднем $O(n * log(n))$
- В худшем $O(n^2)$ - если выбран плохой опорный элемент

---

13. Динамическое программирование. Общие принципы динамического программирования. Восстановление ответа.

Динамическое программирование — способ решения сложных задач путём разбиения их на более простые подзадачи. Он применим к задачам с оптимальной подструктурой, выглядящим как набор перекрывающихся подзадач, сложность которых чуть меньше исходной. В этом случае время вычислений, по сравнению с «наивными» методами, можно значительно сократить.

Метод динамического программирования сверху (мемоизация, ленивая динамика) — это простое запоминание результатов решения тех подзадач, которые могут повторно встретиться в дальнейшем.

Динамическое программирование снизу (табулация) включает в себя переформулирование сложной задачи в виде рекурсивной последовательности более простых подзадач.

Основная идея состоит в том, чтобы

1. свести задачу для к задаче для чисел,меньших, чем N
   (с помощью формулы)
2. хранить все ответы в массиве
3. заполнить начало массива вручную (для которых
   формула не работает)
4. обойти массив и заполнить ответы по формуле
5. вывести ответ откуда-то из этого массива

Восстановление ответа:

1. Хранить дополнительный массив с решением
2. По таблице понять какое решение принято

---

14. Наибольшая общая последовательность

Рассмотрим последовательность чисел $a_1, a_2, \ldots, a_n$. Если вычеркнуть из этой последовательности часть чисел, мы получим другую последовательность, которую называют подпоследовательностью данной последовательности.

Рассмотрим теперь еще одну последовательность $b_1, b_2, \ldots, b_m$. Требуется найти длину самой длинной подпоследовательности последовательности a_i, которая одновременно является и подпоследовательностью последовательности b_j. Такую последовательность называют наибольшей общей подследовательностью (НОП).

Опишем подзадачи, на которые мы будем разбивать нашу задачу. Мы напишем функцию $LCS(p, q)$, которая находит длину НОП для двух начальных участков $a_1, \ldots, a_p$ и $b_1, \ldots, b_q$ наших последовательностей. Пусть для всех пар q и p ($p < n, q < m$), мы задачу решать уже научились. Попробуем вычислить LCS(n, m). Рассмотрим два случая:

1. $a_n = b_m$. Тогда $LCS(n, m) = LCS(n-1, m-1) + 1$.
2. $a_n \neq b_m$. Тогда $LCS(n, m) = max(LCS(n, m-1), LCS(n-1, m))$.

Пользуясь этими формулами, мы можем заполнить таблицу значений $LCS(p, q)$ для всех p и q последовательно: сначала заполняем первую строчку слева направо, затем вторую и т.д. Последнее число в последней строке и будет ответом на поставленную задачу.

```python
def lcs(a, b):
    n = len(a)
    m = len(b)
    table = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if a[i - 1] == b[j - 1]:
                table[i][j] = table[i - 1][j - 1] + 1
            else:
                table[i][j] = max(table[i - 1][j],
                                  table[i][j - 1])
    return table[-1][-1]
```

---

15. Дискретная задача о рюкзаке.

Задача о рюкзаке (Knapsack problem) — дано N предметов, n_i предмет имеет массу $w_i > 0$ и стоимость $p_i > 0$. Необходимо выбрать из этих предметов такой набор, чтобы суммарная масса не превосходила заданной величины W (вместимость рюкзака), а суммарная стоимость была максимальна.

Задачу о рюкзаке можно решить несколькими способами:

- Перебирать все подмножества набора из N предметов. Сложность такого решения $O(2^N)$
- Методом Meet-in-the-middle. Сложность решения $O(2^{N/2}N)$
- Метод динамического программирования. Сложность — $O(NW)$

Для решения построим таблицу размерности N на W, где столбцы соответствуют объему рюкзака, а строки отдельным предметам.

В общем случае формула для стоимости в каждой ячейке выглядит так:

$$S[i, j] = max(S[i - 1, j], p_i + S[i - 1, j - w_i]),$$

где i — номер строки, j — столбца.

Другие вариации задачи:

- **Ограниченный рюкзак** - каждый предмет может быть выбран ограниченное b_i число раз.
- **Неограниченный рюкзак** - любой предмет может быть выбран любое количество раз.
- **Непрерывный рюкзак** - возможно брать любую дробную часть от предмета, при этом удельная стоимость сохраняется.
- и т.д.

16. Редакционное расстояние.

Расстояние Левенштейна (редакционное расстояние или дистанция редактирования) — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую. Нахождения расстояния Левенштейна аналогично нахождению НОП для двух строк, за исключением того, что мы ищем не максимальное, а минимальное количество.

```
def levenshtein_distance(s1, s2):
    n, m = len(s1), len(s2)

    if n > m:
        s1, s2 = s2, s1
        n, m = m, n

    current_row = list(range(n + 1))

    for i in range(1, m + 1):
        previous_row, current_row = current_row, [i] + [0] * n
        for j in range(1, n + 1):
            add = previous_row[j] + 1
            delete = current_row[j - 1] + 1
            change = previous_row[j - 1]
            if s1[j - 1] != s2[i - 1]:
                change += 1
            current_row[j] = min(add, delete, change)

    return current_row[n]
```

17. Односвязный список, двусвязный список.

Линейный однонаправленный список — это структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей. Каждый элемент списка имеет указатель на следующий элемент. Последний элемент списка указывает на None . Элемент, на который нет указателя, является первым (головным) элементом списка. Здесь ссылка в каждом узле указывает на следующий узел в списке. В односвязном списке можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

В информатике линейный список обычно определяется как абстрактный тип данных (АТД), формализующий понятие упорядоченной коллекции данных. На практике линейные списки обычно реализуются при помощи массивов и связных списков.

АТД нетипизированного изменяемого списка может быть определён как набор из конструктора и основных операций:

- Операция, проверяющая список на пустоту.
- Три операции добавления объекта в список (в начало, конец или внутрь после любого (n-го) элемента списка);
- Операция, вычисляющая первый (головной) элемент списка;
- Операция доступа к списку, состоящему из всех элементов исходного списка, кроме первого

Характеристики

- Длина списка.Количество элементов в списке.
- Списки могут быть типизированными и нетипизированными. Если список типизирован, то тип его элементов задан, и все его элементы должны иметь типы, совместимые с заданным типом элементов списка.
- Список может быть сортированным или несортированным.
- В зависимости от реализации может быть возможен произвольный доступ к элементам списка.

Двусвязный список (двунаправленный связный список) - ссылки в каждом узле указывают на предыдущий и на последующий узел в списке.

Как и односвязный список, двусвязный допускает только последовательный доступ к элементам, но при этом дает возможность перемещения в обе стороны.

В этом списке проще производить удаление и перестановку элементов, так как легко доступны адреса тех элементов списка, указатели которых направлены на изменяемый элемент.

Применение связанных списков:

- Для построения более сложных структур данных.
- Для реализации файловых систем.
- Для формирования хэш-таблиц.
- Для выделения памяти в динамических структурах данных.

---

18. Стек. Очередь. Дек.

Стек

Стек (stack — стопка) — структура данных, представляющая из себя упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека. Притом первым из стека удаляется элемент, который был помещен туда последним, то есть в стеке реализуется стратегия «последним вошел — первым вышел» (last-in, first-out — LIFO).

Операции стека:

- `empty` — проверка стека на наличие в нем элементов,
- `push` (запись в стек) — операция вставки нового элемента,
- `pop` (снятие со стека) — операция удаления нового элемента.

Применение стека

- Для реализации рекурсии.
- Для вычислений постфиксных значений.
- Для временного хранения данных, например истории запросов или изменений.

Очередь

Очередь (queue) — это структура данных, добавление и удаление элементов в которой происходит путём операций push и pop соответственно. Притом первым из очереди удаляется элемент, который был помещен туда первым, то есть в очереди реализуется принцип «первым вошел — первым вышел» (first-in, first-out — FIFO). У очереди имеется голова (head) и хвост (tail). Когда элемент ставится в очередь, он занимает место в её хвосте. Из очереди всегда выводится элемент, который находится в ее голове.

Очередь поддерживает следующие операции:

- `empty` — проверка очереди на наличие в ней элементов,
- `push` (запись в очередь) — операция вставки нового элемента,
- `pop` (снятие с очереди) — операция удаления нового элемента,
- `size` — операция получения количества элементов в очереди.

Применение очереди

- Для реализации очередей, например на доступ к определённому ресурсу.
- Для управления потоками в многопоточных средах.
- Для генерации значений.
- Для создания буферов.

Дек

Дек (deque — double ended queue) — структура данных, представляющая из себя список элементов, в которой добавление новых элементов и удаление существующих производится с обоих концов. Эта структура поддерживает как FIFO, так и LIFO, поэтому на ней можно реализовать как стек, так и очередь. Дек можно воспринимать как двустороннюю очередь.

Дек имеет следующие операции:

- `empty` — проверка на наличие элементов,
- `pushBack` (запись в конец) — операция вставки нового элемента в конец,
- `popBack` (снятие с конца) — операция удаления конечного элемента,
- `pushFront` (запись в начало) — операция вставки нового элемента в начало,
- `popFront` (снятие с начала) — операция удаления начального элемента.

---

19. Куча

Куча (heap) — это полное двоичное дерево, удовлетворяющее свойству кучи: если узел A — это родитель узла B, то ключ узла A больше либо равен ключу узла B.

- Если любой узел всегда больше дочернего узла (узлов), а ключ корневого узла является наибольшим среди всех остальных узлов, это **max-куча**.
- Если любой узел всегда меньше дочернего узла (узлов), а ключ корневого узла является наименьшим среди всех остальных узлов, это **min-куча**.

Применяется для:

- Пирамидальной сортировки
- Алгоритмы поиска
- Алгоритмы на графах (Дейкстры, Прима)
- Очереди с приоритетом

Полная и почти полная бинарная куча может быть представлена очень эффективным способом с помощью индексного массива. Первый (или последний) элемент будет содержать корень. Следующие два элемента массива содержат узлы-потомки корня. Следующие четыре элемента содержат четверых потомков от двух узлов — потомков корня, и так далее. Таким образом, потомки узла уровня n будут расположены на позициях `2n` и `2n+1` для массива, индексируемого с единицы, или на позициях `2n+1` и `2n+2` для массива, индексируемого с нуля.

Операция	**найти мин.**	**добавить**	**удалить мин.**	**уменьшить ключ**	**слияние**
Асимптотика	O(1)	O(log n)	O(log n)	O(log n)	O(n)

20. Графы. Способы хранения: матрица смежности, списки смежности, матрица инцидентности.

Граф — математическая абстракция реальной системы любой природы, объекты которой обладают парными связями.

Граф как математический объект есть совокупность двух множеств — множества самих объектов, называемого множеством вершин, и множества их парных связей, называемого множеством рёбер. Элемент множества рёбер есть пара элементов множества вершин.

Простой (неориентированный) граф G(V, E) есть совокупность двух множеств – непустого множества вершин и множества рёбер - неупорядоченных пар различных элементов множества вершин.

Ориентированный граф (орграф) есть совокупность двух множеств – непустого множества вершин и множества дуг или упорядоченных пар различных элементов множества вершин.

Взвешенный граф — граф, каждому ребру которого поставлено в соответствие некое значение (вес ребра).

Способы хранения

Матрица смежности

Матрицей смежности $A = ||\alpha_{i,j}||$ невзвешенного графа $G = (V, E)$ называется матрица $A[V \times V]$, в которой $\alpha_{i,j}$ — количество рёбер, соединяющих вершины v_i и v_j, причём при $i = j$ каждую петлю учитываем дважды, если граф не является ориентированным, и один раз, если граф ориентирован.

Матрицей смежности $A = ||\alpha_{i,j}||$ взвешенного графа $G = (V, E)$ называется матрица $A[V \times V]$, в которой $\alpha_{i,j}$ — вес ребра, соединяющего вершины v_i и v_j.

Университет Сириус Колледж **Матрица смежности**

Плюсы:

- Добавление ребра, удаление ребра, проверка наличия ребра между вершинами i и j за $O(n)$
- Лучший выбор для плотных графов. В случае разреженного графа и матрицы смежности можно использовать структуры данных для разреженных матриц
- Возможность выполнения операция на GPU

Минусы:

- Требуется VxV памяти для хранения. Чаще всего графы не имеют большого количества связей и лучшим выбором будут списки смежности.
- Выполнения операций по нахождению внешних и внутренних ребер требует большего времени

Список смежности

Список смежности — один из способов представления графа в виде коллекции списков вершин. Каждой вершине графа соответствует список, состоящий из «соседей» этой вершины.

Плюсы:

- Эффективны в плане потребления памяти, так как хранится только информация о ребрах. Для больших разреженных графов могут сберечь большой объем памяти
- Быстрый поиск смежных вершин

Минусы:

- Построение списка смежности не быстрее построения матрицы смежности, так как необходимо так же проверить и найти все узлы

Матрица инцидентности

Вершина v **инцидентна** ребру e, если $v \in e$; тогда еще говорят, что e есть ребро при v;

Матрицей инцидентности (инциденций) неориентированного графа называется матрица $I(|V| \times |E|)$, для которой $I_{i,j} = 1$, если вершина v_i инцидентна ребру e_j, в противном случае $I_{i,j} = 0$.

Матрицей инцидентности (инциденций) ориентированного графа называется матрица $I(|V| \times |E|)$, для которой $I_{i,j} = 1$, если вершина v_i является началом дуги e_j, $I_{i,j} = -1$, если v_i является концом дуги e_j, в остальных случаях $I_{i,j} = 0$.

Сравнение списка смежности и матрицы смежности (возможно, необязательно)

Операция	Список смежности	Матрица смежности
Проверка на наличие ребра (x, y)	$O(\|V\|)$	$O(1)$
Определение степени вершины	$O(1)$	$O(\|V\|)$
Использование памяти для разреженных графов	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$
Вставка/удаление грани	$O(1)$	$O(d)$
Обход графа	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$

---

21. Поиск в глубину в неориентированных графах.

Обход в глубину (поиск в глубину, Depth-First Search, DFS) — один из основных методов обхода графа, часто используемый для проверки связности, поиска цикла и компонент сильной связности, а так же для топологической сортировки.

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

Пошаговое представление:

1. Выбираем любую вершину из еще не пройденных, обозначим ее как `u`.
2. Запускаем процедуру `dfs(u)`
    1. Помечаем вершину `u` как пройденную
    2. Для каждой не пройденной смежной с `u` вершиной (назовем ее `v`) запускаем `dfs(v)`

3. Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

Время работы алгоритма оценивается как $O(V + E)$.

---

22. Выделение компонент связности.

Компонента связности - набор вершин графа, между любой парой которых существует путь.

Для поиска компонент связности используется обычный DFS практически без модификаций. При запуске обхода из одной вершины, он гарантированно посетит все вершины, до которых возможно добраться, то есть, всю компоненту связности, к которой принадлежит начальная вершина. Для нахождения всех компонент просто попытаемся запустить обход из каждой вершины по очереди, если мы ещё не обошли её компоненту ранее.

Простейший вариант: просто заполнить список `comp`, где `comp[i]` - номер компоненты связности, к которой принадлежит вершина `i`.

---

23. Поиск в глубину в ориентированных графах: ориентированные ациклические графы.

Ориентированный ациклический граф (направленный ациклический граф, DAG, directed acyclic graph) — орграф, в котором отсутствуют направленные циклы, но могут быть «параллельные» пути, выходящие из одного узла и разными путями приходящие в конечный узел. Направленный ациклический граф является обобщением дерева (точнее, их объединения — леса).

Направленные ациклические графы широко используются в приложениях: в компиляторах, в искусственном интеллекте (для представления искусственных нейронных сетей без обратной связи), в статистике и машинном обучении.

Нахождение цикла в орграфе

1. Пометить текущий узел как посещенный и добавить его индекс в стек.
2. Пройти в цикле по вершинам, выполнить рекурсивный вызов функции `dfs` (данный шаг необходим для гарантии, что, если граф является лесом, мы проверим все подграфы):

    1. В каждом рекурсивном вызове найти все смежные вершины для данной вершины:
        1. Если смежная вершина уже добавлена в стек, то граф циклический, возвращаем истину.
        2. Иначе, вызываем рекурсивную функцию для смежной вершины.
    2. При выходе из рекурсивного вызова, удалить текущий узел из стека, чтобы показать, что он более не является частью проверяемого пути.
3. Если какая либо из функций возвращает истину, остановить дальнейшее выполнение и вернуть истину в качестве результата.

---

24. Топологическая сортировка вершин

Топологическая сортировка для ориентированного ациклического графа (Directed Acyclic Graphs, DAG) — это линейное упорядочение вершин, для которого выполняется следующее условие — для каждого направленного ребра uv вершина u предшествует вершине v в упорядочении. Если граф не является DAG, то топологическая сортировка для него невозможна.

Алгоритм Кана

1. Вычислить количество входящих дуг для каждой вершины в графе и установить начальное значение счетчика посещенных узлов на 0.
2. Выбрать все вершины с 0 входящих дуг и поставить их все в очередь.
3. Добавить одну посещенную вершину к счетчику после удаления вершины из очереди.
4. Для каждой смежной вершины уменьшить число входов на 1.
5. Добавить вершину в очередь, если если число входов любой из смежных вершин уменьшилось до 0.
6. Пока очередь не пуста, повторять с шага 3.
7. Топологическая сортировка невозможна для графа если число посещенных узлов не равно числу вершин графа.

---

25. Нахождение кратчайших путей из одной вершины в невзвешенных графах, поиск в ширину.

Задача о кратчайшем пути — задача поиска самого короткого пути между двумя вершинами на графе, в которой минимизируется сумма весов рёбер, составляющих

путь.

Дан невзвешенный ориентированный граф $G = (V, E)$, а также вершина s. Найти длину кратчайшего пути от s до каждой из вершин графа. Длина пути — количество рёбер в нём.

Обход в ширину (Поиск в ширину, BFS, Breadth-first search) — один из простейших алгоритмов обхода графа, являющийся основой для многих важных алгоритмов для работы с графами.

Алгоритм работает следующим образом.

1. Создадим массив $dist$ расстояний. Изначально $dist[s] = 0$ (поскольку расстояний от вершины до самой себя равно 0) и $dist[v] = \infty$ для $v \neq s$.
2. Создадим очередь q. Изначально в q добавим вершину s.
3. Пока очередь q не пуста, делаем следующее:
    1. Извлекаем вершину v из очереди.
    2. Рассматриваем все рёбра $(v, u) \in E$. Для каждого такого ребра пытаемся сделать релаксацию: если $dist[v] + 1 < dist[u]$, то мы делаем присвоение $dist[u] = dist[v] + 1$ и добавляем вершину u в очередь.

---

26. Нахождение кратчайших путей из одной вершины в графах с положительными весами.

> *P.S. Странный билет какой-то, непонятно, что нужно тут*

Дан взвешенный ориентированный граф $G = (V, E)$, а также вершина s. Длина ребра (u, v) равна $w(u, v)$. Длины всех рёбер неотрицательные. Найти длину кратчайшего пути от s до каждой из вершин графа. Длина пути — сумма длин рёбер в нём.

Алгоритм Дейкстры (Dijkstra's algorithm) — алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании, например, его используют протоколы маршрутизации OSPF и IS-IS.

Кратчайшие пути во взвешенных графах. Алгоритм Дейкстры

1. Создать массив $dist$ расстояний. Изначально $dist[s] = 0$ и $dist[v] = \infty$ для $v \neq s$.

2. Создать булев массив $used$, $used[v] = 0$ для всех вершин v — в нём мы будем отмечать, совершалась ли релаксация из вершины.
3. Пока существует вершина v такая, что $used[v] = 0$ и $dist[v] \neq \infty$, притом, если таких вершин несколько, то v — вершина с минимальным $dist[v]$, делать следующее:
    1. Пометить, что мы совершали релаксацию из вершины v, то есть присвоить $used[v] = 1$.
    2. Рассматриваем все рёбра $(v, u) \in E$. Для каждого ребра пытаемся сделать релаксацию: если $dist[v] + w(v, u) < dist[u]$, присвоить $dist[u] = dist[v] + w(v, u)$.

---

27. Алгоритм Дейкстры, оценка времени работы при различных реализациях очереди с приоритетами (массивом, двоичной кучей).

Алгоритм Дейкстры (Dijkstra’s algorithm) — алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании, например, его используют протоколы маршрутизации OSPF и IS-IS.

Кратчайшие пути во взвешенных графах. Алгоритм Дейкстры

1. Создать массив $dist$ расстояний. Изначально $dist[s] = 0$ и $dist[v] = \infty$ для $v \neq s$.
2. Создать булев массив $used$, $used[v] = 0$ для всех вершин v — в нём мы будем отмечать, совершалась ли релаксация из вершины.
3. Пока существует вершина v такая, что $used[v] = 0$ и $dist[v] \neq \infty$, притом, если таких вершин несколько, то v — вершина с минимальным $dist[v]$, делать следующее:
    1. Пометить, что мы совершали релаксацию из вершины v, то есть присвоить $used[v] = 1$.
    2. Рассматриваем все рёбра $(v, u) \in E$. Для каждого ребра пытаемся сделать релаксацию: если $dist[v] + w(v, u) < dist[u]$, присвоить $dist[u] = dist[v] + w(v, u)$.

Вычислим время работы алгоритма. Мы V раз ищем вершину минимальным $dist$, поиск минимума линейный за $O(V)$, отсюда $O(V^2)$. Обработка ребер происходит суммарно за $O(E)$, потому что на каждое ребро мы тратим $O(1)$ действий.
Таким образом, финальная асимптотика: $O(V^2 + E)$.

Искать вершину с минимальным $dist$ можно гораздо быстрее, используя такую структуру данных как очередь с приоритетом.

Искать вершину с минимальным $dist$ можно гораздо быстрее, используя такую структуру данных как очередь с приоритетом. Нам нужно хранить пары $(dist, index)$ и уметь делать такие операции:

- Извлечь минимум (чтобы обработать новую вершину)
- Удалить вершину по индексу (чтобы уменьшить $dist$ до какого-то соседа)
- Добавить новую вершину (чтобы уменьшить $dist$ до какого-то соседа)

Для этого используют, например, кучу или сет. Удобно помимо сета хранить сам массив dist, который его дублирует, но хранит элементы по порядку. Тогда, чтобы заменить значение $(dist1, u)$ на $(dist2, u)$, нужно удалить из сета значение $(dist[u], u)$, сделать $dist[u] = dist2$; и добавить в сет $(dist[u], u)$.

Данный алгоритм будет работать за $O(\log V)$ извлечений минимума и $O(E \log V)$ операций уменьшения расстояния до вершины. Поэтому алгоритм работает за $O(E \log V)$.

Заметьте, что этот алгоритм не лучше и не хуже алгоритма без сета, который работает за $O(V^2 + E)$. Ведь если $E = O(V^2)$ (граф почти полный), то Дейкстра без сета работает быстрее, а если, например, $E = O(V)$, то Дейкстра на сете работает быстрее.

28. Кратчайшие пути в ациклических ориентированных графах.

Пусть дан ациклический ориентированный взвешенный граф. Требуется найти вес кратчайшего пути из u в v.

Пусть d — функция, где $d(i)$ — вес кратчайшего пути из u в i. Ясно, что $d(u)$ равен 0. Пусть $w(i, j)$ — вес ребра из i в j. Будем обходить граф в порядке топологической сортировки. Получаем следующие соотношения:

$$d(i) = \min\left(d(i), d(j) + w(j, i)\right)$$

Так как мы обходим граф в порядке топологической сортировки, то на i-ом шаге всем $d(j)$ (j такие, что существует ребро из j в i) уже присвоены оптимальные ответы, и, следовательно, $d(i)$ также будет присвоен оптимальный ответ.

29. Алгоритм Беллмана-Форда, проверка наличия цикла отрицательного веса.

Алгоритм Беллмана-Форда предназначен для решения задачи поиска кратчайшего пути на графе. Для заданного ориентированного взвешенного графа алгоритм находит

кратчайшие расстояния от выделенной вершины-источника до всех остальных вершин графа.

Алгоритм Беллмана-Форда масштабируется хуже других алгоритмов решения указанной задачи (сложность $O(|V||E|)$ против $O(|E| + |V| \ln |V|)$ у алгоритма Дейкстры), однако его отличительной особенностью является применимость к графам с произвольными, в том числе отрицательными, весами.

1. Инициализация: всем вершинам присваивается предполагаемое расстояние $dist[v] = \infty$, кроме вершины-источника, для которой $dist(u) = 0$.
2. Релаксация множества рёбер E
    1. Для каждого ребра $e = (v, z) \in E$ вычисляется новое предполагаемое расстояние $new_dist(z) = dist(v) + w(e)$.
    2. Если $new_dist(z) < dist(z)$, то происходит присваивание $dist(z) = new_dist(z)$ (релаксация ребра e).
3. Алгоритм производит релаксацию всех рёбер графа до тех пор, пока на очередной итерации происходит релаксация хотя бы одного ребра.

Проверка наличия цикла отрицательного веса

Алгоритм Форда-Беллмана сможет бесконечно делать релаксации среди всех вершин этого цикла и вершин, достижимых из него. Следовательно, если не ограничивать число фаз числом $n - 1$, то алгоритм будет работать бесконечно, постоянно улучшая расстояния до этих вершин.

Отсюда мы получаем **критерий наличия достижимого цикла отрицательного веса**: если после $n - 1$ фазы мы выполним ещё одну фазу, и на ней произойдёт хотя бы одна релаксация, то граф содержит цикл отрицательного веса, достижимый из v; в противном случае, такого цикла нет.

---

30. Кратчайшие пути между всеми парами вершин: алгоритм Флойда-Уоршелла.

Алгоритм Флойда (алгоритм Флойда–Уоршелла) — алгоритм нахождения длин кратчайших путей между всеми парами вершин во взвешенном ориентированном графе.

Работает корректно, если в графе нет циклов отрицательной величины, а в случае, когда такой цикл есть, позволяет найти хотя бы один такой цикл. Алгоритм работает за $O(n^3)$ времени и использует $O(n^2)$ памяти. Разработан в 1962 году.

Дан взвешенный ориентированный граф $G(V, E)$, в котором вершины пронумерованы от 1 до n. Требуется найти матрицу кратчайших расстояний d, в которой элемент d_{ij}

либо равен длине кратчайшего пути из i в j, либо равен $+\infty$, если вершина j не достижима из i.

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

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

---

31. Деревья. Бинарные деревья.

Дерево — это связный ациклический граф. Связность означает наличие маршрута между любой парой вершин, ацикличность — отсутствие циклов. Отсюда, в частности, следует, что число рёбер в дереве на единицу меньше числа вершин, а между любыми парами вершин имеется один и только один путь.

Ориентированное (направленное) дерево — ацикличный орграф, в котором только одна вершина имеет нулевую степень захода (в неё не ведут дуги), а все остальные вершины имеют степень захода 1 (в них ведёт ровно по одной дуге). Вершина с нулевой степенью захода называется корнем дерева, вершины с нулевой степенью исхода (из которых не исходит ни одна дуга) называются концевыми вершинами или листьями.

Двоичное дерево (Бинарное дерево) — иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками. Двоичное дерево является упорядоченным ориентированным деревом.

Для практических целей обычно используют два подвида двоичных деревьев — двоичное дерево поиска и двоичная куча.

Бинарное дерево можно представить в виде списка (Потомками узла `n` будут элементы с индексами `2n + 1` и `2n + 2`) или отдельным классом:

```
class Node:
    def __init__(self, data, left=None, right=None):
        self.left = left
```

```
        self.right = right
        self.data = data
```

Основные операции с бинарным деревом:

- Вставка элемента
- Удаление элемента
- Поиск элемента
- Удаление элемента по значению
- Обход дерева

Дополнительные операции с бинарным деревом:

- Нахождение высоты дерева
- Нахождение слоя дерева
- Нахождение размера дерева

Применение

- В компиляторах, в частности для выполнения арифметических выражений
- Деревья кодирования Хаффмана в алгоритмах сжатия
- Очереди с приоритетом
- Представления иерархических данных
- В табличных редакторах
- Для индексирования баз данных и кэша
- Для быстрого поиска
- Выделения памяти в компьютерах
- Операций кодирования и декодирования
- Для получения и организации информации из больших объемов данных
- В моделях принятия решений
- В алгоритмах сортировки

Обход дерева:

- Обход в глубину (DFS)
  - Прямой обход - Префиксный - Корень-Левое-Правое
  - Центрированный обход - Инфиксный - Левое-Корень-Правое
  - Обратный обход - Постфиксный - Левое-Правое-Корень
- Обход в ширину (BFS) - Уровневый обход

---

32. Деревья поиска.

Бинарное дерево поиска (binary search tree, BST) — структура данных для работы с упорядоченными множествами.

Бинарное дерево поиска обладает следующим свойством: если x — узел бинарного дерева с ключом k, то все узлы в левом поддереве должны иметь ключи, меньшие k, а в правом поддереве большие k.

Основным преимуществом двоичного дерева поиска перед другими структурами данных является возможная высокая эффективность реализации основанных на нём алгоритмов поиска и сортировки.

Применение бинарных деревьев поиска

- Для индексации
- Для реализации алгоритмов поиска
- Для реализации других структур данных
- Как часть систем принятия решения, компьютерных симуляций для хранения и быстрого доступа к данным
- Для реализации систем автодополнения и проверки орфографии

Преимущества:

- Высокая скорость вставки и удаления на сбалансированном дереве ($O \log n$))
- Высокая скорость поиска ($O(\log n)$)
- Эффективное использование памяти
- Позволяют искать значения из диапазона
- Простая реализация
- Автоматическая сортировка элементов при добавлении

Недостатки:

- Всегда необходимо реализовывать сбалансированное бинарное дерево поиска, иначе дерево может выродиться в список, что увеличивает время выполнения операций
- Плохо подходят для случайного доступа к элементам
- Не поддерживают некоторые операции

---

33. Красно-чёрные деревья

Красно-чёрное дерево (англ. red-black tree) — двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" (англ. red) и "чёрный" (англ. black).

При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными.

Для экономии памяти фиктивные листья можно сделать одним общим фиктивным листом.

1. Каждый узел промаркирован красным или чёрным цветом
2. Корень и конечные узлы (листья) дерева — чёрные
3. У красного узла родительский узел — чёрный
4. Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов
5. Чёрный узел может иметь чёрного родителя

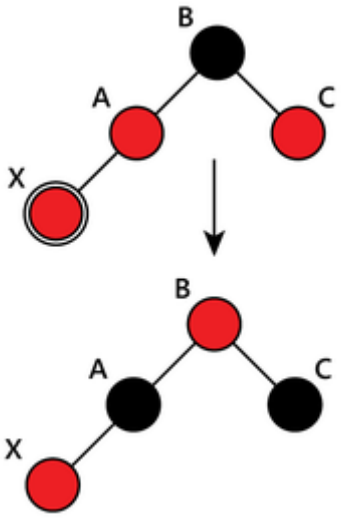
Вставка

Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего сына не станет `None` (то есть этот сын — лист). Вставляем вместо него новый элемент с нулевыми потомками и красным цветом. Теперь проверяем балансировку.

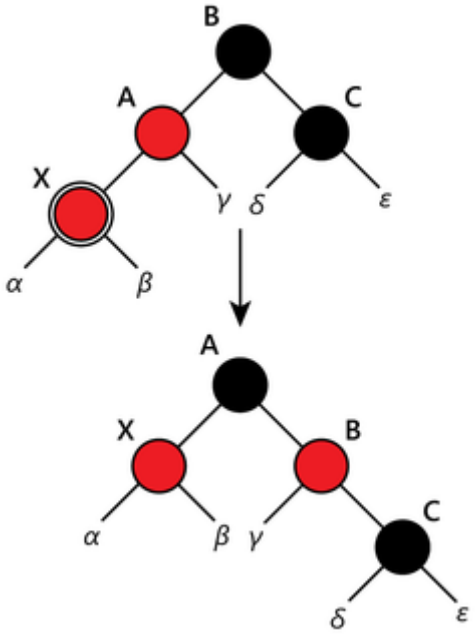
Если отец нового элемента черный, то никакое из свойств дерева не нарушено.

Если же он красный, то нарушается свойство 3, для исправления достаточно рассмотреть два случая:

1. "Дядя" этого узла тоже красный. Тогда, чтобы сохранить свойства 3 и 4, просто перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" — в красный.
   В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин "отцы" черные.
   Проверяем, не нарушена ли балансировка. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет, чтобы дерево удовлетворяло свойству 2



2. "Дядя" чёрный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот относительно деда.
   Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком. Таким образом, свойство 3 и постоянство черной высоты сохраняются.



Удаление

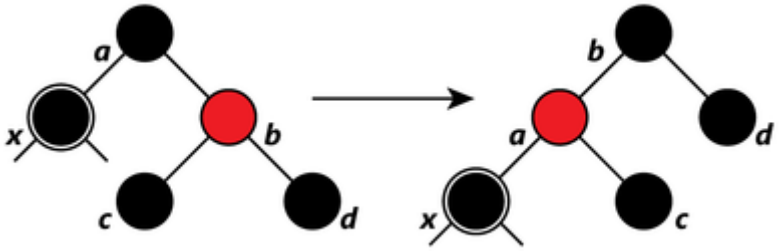
При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

1. Если у вершины нет детей, то изменяем указатель на неё у родителя на `None`.
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.

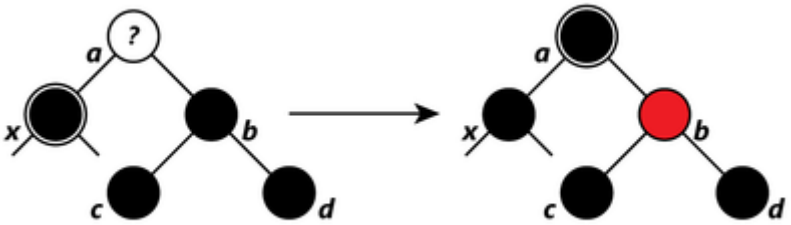
Проверим балансировку дерева.

- Так как при удалении красной вершины свойства дерева не нарушаются.
- Восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

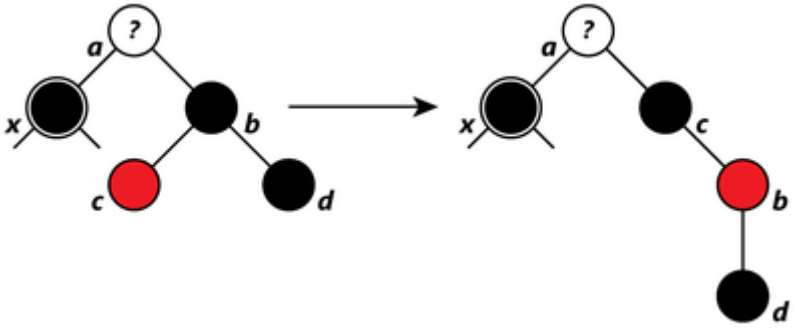
1. Брат красный:
   Делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас x имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.



2. 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 влиянние удаленного чёрного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.

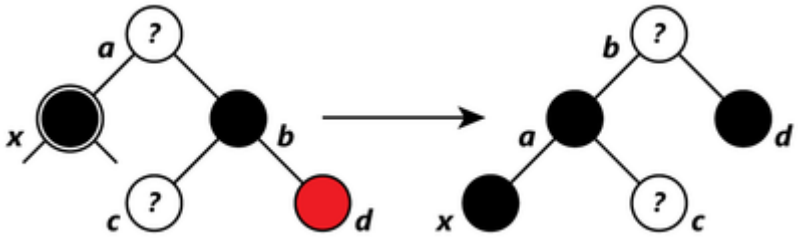


3. Если у брата правый ребёнок чёрный, а левый красный,
   Перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x, ни его отец не влияют на эту трансформацию.



4. Если у брата правый ребёнок красный
   Перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 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.



Преимущества

1. Самое главное преимущество красно-черных деревьев в том, что при вставке выполняется не более $O(1)$ вращений.
2. Процедуру балансировки практически всегда можно выполнять параллельно с процедурами поиска, так как алгоритм поиска не зависит от атрибута цвета узлов.
3. Сбалансированность этих деревьев хуже, чем у АВЛ, но работа по поддержанию сбалансированности в красно-чёрных деревьях обычно эффективнее. Для балансировки красно-чёрного дерева производится минимальная работа по сравнению с АВЛ-деревьями.
4. Использует всего 1 бит дополнительной памяти для хранения цвета вершины.

Красно-чёрные деревья являются наиболее активно используемыми на практике самобалансирующимися деревьями поиска. В частности, ассоциативные контейнеры библиотеки STL(map, set, multiset, multimap) основаны на красно-чёрных деревьях. TreeMap в Java тоже реализован на основе красно-чёрных деревьев.

---

34. AVL-деревья.

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

АВЛ-деревья названы по первым буквам фамилий их изобретателей, Г. М. Адельсона-Вельского и Е. М. Ландиса, которые впервые предложили использовать АВЛ-деревья в 1962 году.

```
class Node:
    def __init__(self, val):
        self.data = val
        self.left = None
```

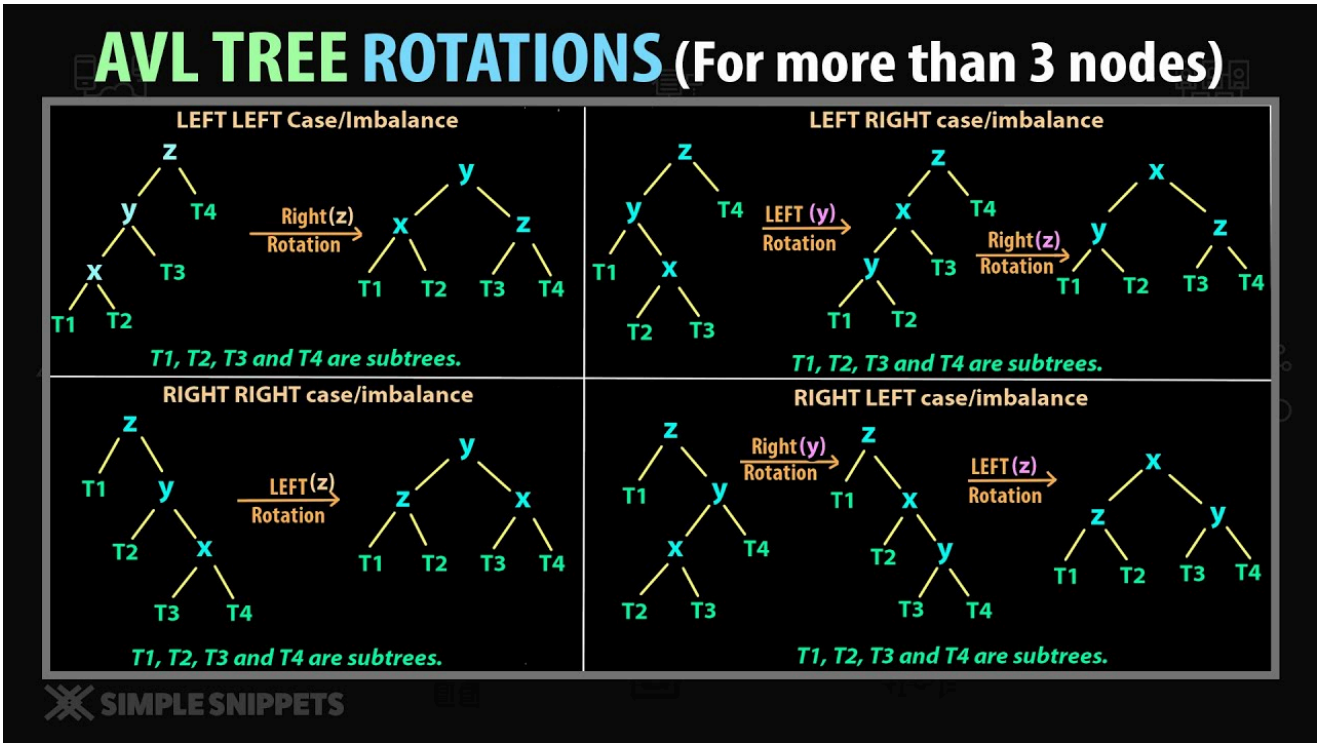
```
        self.right = None
        self.height = 1
```

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 $|h(L) - h(R)| \leqslant 1$, иначе ничего не меняет. Для балансировки будем хранить для каждой вершины разницу между высотой её левого и правого поддерева `diff[i]=h(L)-h(R)`

Для балансировки вершины используются один из 4 типов вращений:

- Малое левое вращение
- Большое левое вращение (Правое-левое)
- Малое правое вращение
- Большое правое вращение (Левое правое)



Добавление вершины

Пусть нам надо добавить ключ t. Будем спускаться по дереву, как при поиске ключа t. Если мы стоим в вершине a и нам надо идти в поддерево, которого нет, то делаем ключ t листом, а вершину a его корнем. Дальше поднимаемся вверх по пути поиска и пересчитываем баланс у вершин.

Если мы поднялись в вершину i из левого поддерева, то `diff[i]` увеличивается на единицу, если из правого, то уменьшается на единицу.

Если пришли в вершину и её баланс стал равным нулю, то это значит высота поддерева не изменилась и подъём останавливается. Если пришли в вершину и её баланс стал равным 1 или -1, то это значит высота поддерева изменилась и подъём продолжается.

Если пришли в вершину и её баланс стал равным 2 или -2, то делаем одно из четырёх вращений и, если после вращения баланс стал равным нулю, то останавливаемся, иначе продолжаем подъём.

Так как в процессе добавления вершины мы рассматриваем не более, чем $O(h)$ вершин дерева, и для каждой запускаем балансировку не более одного раза, то суммарное количество операций при включении новой вершины в дерево составляет $O(\log n)$ операций.

Удаление вершины

Для простоты опишем рекурсивный алгоритм удаления. Если вершина — лист, то удалим её, иначе найдём самую близкую по значению вершину a, переместим её на место удаляемой вершины и удалим вершину a.

От удалённой вершины будем подниматься вверх к корню и пересчитывать баланс у вершин.

Если мы поднялись в вершину i из левого поддерева, то `diff[i]` уменьшается на единицу, если из правого, то увеличивается на единицу.

Если пришли в вершину и её баланс стал равным 1 или -1, то это значит, что высота этого поддерева не изменилась и подъём можно остановить. Если баланс вершины стал равным нулю, то высота поддерева уменьшилась и подъём нужно продолжить.

Если баланс стал равным 2 или -2, следует выполнить одно из четырёх вращений и, если после вращений баланс вершины стал равным нулю, то подъём продолжается, иначе останавливается.

В результате указанных действий на удаление вершины и балансировку суммарно тратится, как и ранее, $O(h)$ операций. Таким образом, требуемое количество действий — $O(\log n)$.

---

35. B деревья.

B-дерево — структура данных, дерево поиска. С точки зрения внешнего логического представления — сбалансированное, сильно ветвистое дерево. Часто используется для хранения данных во внешней памяти.

Использование B-деревьев впервые было предложено Р. Бэйером и Э. МакКрейтом в 1970 году.

С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц памяти, то есть каждому узлу дерева соответствует блок памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.

Структура B-дерева применяется для организации индексов во многих современных СУБД.

Свойства

1. Ключи в каждом узле обычно упорядочены для быстрого доступа к ним. Корень содержит от 1 до $2t-1$ ключей. Любой другой узел содержит от $t-1$ до $2t-1$ ключей. Листья не являются исключением из этого правила. Здесь t — параметр дерева, не меньший 2 (и обычно принимающий значения от 50 до 2000).
2. У листьев потомков нет. Любой другой узел, содержащий ключи $K_1, \ldots, K_n$, содержит $n+1$ потомков. При этом
    1. Первый потомок и все его потомки содержат ключи из интервала $(-\infty, K_1)$
    2. Для $2 \leq i \leq n$, i-й потомок и все его потомки содержат ключи из интервала (K_{i-1}, K_i)
    3. $(n+1)$-й потомок и все его потомки содержат ключи из интервала (K_n, ∞)
3. Глубина всех листьев одинакова.

Свойство 2 можно сформулировать иначе: каждый узел B-дерева, кроме листьев, можно рассматривать как упорядоченный список, в котором чередуются ключи и указатели на потомков.

- Во всех случаях полезное использование пространства вторичной памяти составляет свыше 50 %. С ростом степени полезного использования памяти не происходит снижения качества обслуживания.
- Произвольный доступ к записи реализуется посредством малого количества подопераций (обращения к физическим блокам).
- В среднем достаточно эффективно реализуются операции включения и удаления записей; при этом сохраняется естественный порядок ключей с целью последовательной обработки, а также соответствующий баланс дерева для обеспечения быстрой произвольной выборки.
- Неизменная упорядоченность по ключу обеспечивает возможность эффективной пакетной обработки.

Основной недостаток B-деревьев состоит в отсутствии для них эффективных средств выборки данных (то есть метода обхода дерева), упорядоченных по свойству, отличному от выбранного ключа.

Вставка

1. Если дерево пустое, добавить корень и вставить значение.
2. Обновить количество ключей в узле.
3. Найти подходящий для вставки узел.
4. Если узел полон, то:
    1. Вставить элемент в порядке возрастания.
    2. Так как количество элементов больше предела, разбить узел по медиане.
    3. Сместить медианный ключ вверх и сделать ключи слева левым потомком, а ключи справа - правым.
5. Если узел не полон, то:
    1. Вставить элемент в порядке возрастания.

---

36. B+ деревья.

B+ дерево — структура данных на основе B-дерева, сбалансированное n-арное дерево поиска с переменным, но зачастую большим количеством потомков в узле. B+-дерево состоит из корня, внутренних узлов и листьев, корень может быть либо листом, либо узлом с двумя и более потомками.

Изначально структура предназначалась для хранения данных в целях эффективного поиска в блочно-ориентированной среде хранения — в частности, для файловых систем; применение связано с тем, что в отличие от бинарных деревьев поиска, B+ деревья имеют очень высокий коэффициент ветвления (число указателей из родительского узла на дочерние — обычно порядка 100 или более), что снижает количество операций ввода-вывода, требующих поиска элемента в дереве.

Вариант B+ дерева, в котором все значения сохранялись в листовых узлах, систематически рассмотрен в 1979 году, притом отмечено, что такие структуры использовались IBM в технологии файлового доступа для мейнфреймов VSAM по крайней мере с 1973 года.

Структура широко применяется в файловых системах — NTFS, ReiserFS, NSS, XFS, JFS, ReFS и BFS используют этот тип дерева для индексирования метаданных; BeFS также использует B+-деревья для хранения каталогов. Реляционные системы управления базами данных, такие как DB2, Informix, Microsoft SQL Server, Oracle Database (начиная с версии 8), Adaptive Server Enterprise и SQLite поддерживают этот тип деревьев для табличных индексов. Среди NoSQL-СУБД, работающих с моделью «ключ—значение», структура данных реализована для доступа к данным в CouchDB, MongoDB (при использовании подсистемы хранения WiredTiger) и Tokyo Cabinet.

Свойства

1. Ключи в каждом узле обычно упорядочены для быстрого доступа к ним. Корень содержит от 1 до $2t - 1$ ключей. Любой другой узел содержит от $t - 1$ до $2t - 1$ ключей. Листья не являются исключением из этого правила. Здесь t — параметр дерева, не меньший 2 (и обычно принимающий значения от 50 до 2000).
2. У листьев потомков нет. Любой другой узел, содержащий ключи $K_1, \ldots, K_n$, содержит $n + 1$ потомков. При этом
    1. Первый потомок и все его потомки содержат ключи из интервала $(-\infty, K_1)$
    2. Для $2 \leq i \leq n$, i-й потомок и все его потомки содержат ключи из интервала (K_{i-1}, K_i)
    3. $(n + 1)$-й потомок и все его потомки содержат ключи из интервала (K_n, ∞)
3. Глубина всех листьев одинакова.
4. Листья имеют ссылку на соседа, позволяющую быстро обходить дерево в порядке возрастания ключей, и ссылки на данные.

```
class Node:
    def __init__(self, order):
        self.order = order
        self.values = []
        self.keys = []
        self.nextKey = None
        self.parent = None
        self.check_leaf = False
```

---

37. Поиск, вставка, удаление, поиск следующего и предыдущего элемента за время, пропорциональное высоте

Поиск элемента

```
def search(node: Node, key):
    if not node or node.data == key:
        return node

    if node.data > key:
        return search(node.left, key)

    return search(node.right, key)
```

Поиск след. или пред. элемента

Найти элемент со следующим значением ключа, относительно ключа некоторого узла.

```
def searchNext(node, key):
    if not node:
        return None

    if node.data == key:
        if node.right:
            current = node.right
            while current.left:
                current = current.left
            return current
        return None

    if node.data > key:
        return searchNext(node.left, key)
    return searchNext(node.right, key)
```

Поиск предыдущего элемента осуществляется аналогично.

Вставка элемента

```
def insert(node, key):
    if node is None:
        return Node(key)

    if key < node.data:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)

    return node
```

Удаление элемента

```
def deleteNode(root, key):
    if root is None:
        return root

    if key < root.data:
        root.left = deleteNode(root.left, key)
    elif(key > root.data):
        root.right = deleteNode(root.right, key)
    else:
        if root.left is None:
            return root.right

        if root.right is None:
            return root.left
```

```
        temp = searchNext(root, key)
        root.data = temp.data
        root.right = deleteNode(root.right, temp.data)

    return root
```